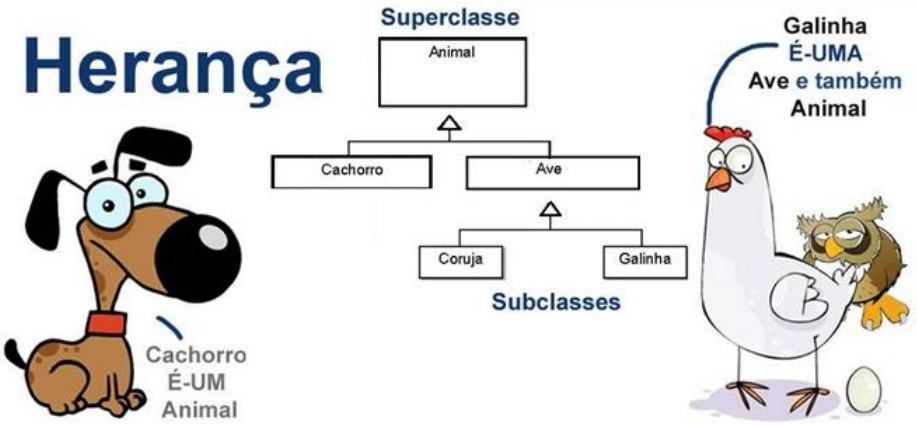
# UNIDADE 3 – HERANÇA, POLIMORFISMO E CLASSES ABSTRATAS

## MÓDULO 1 – HERANÇA

01

## 1 - ENTENDENDO A HERANÇA

Herança significa o legado, na maioria das vezes, bens materiais que são deixados para uma pessoa quando outra pessoa parte dessa vida. Também estamos acostumados a usar esse termo para identificar a herança genética, ou seja, os genes que os nossos pais nos passaram e os quais determinam que tenhamos traços e características dos nossos pais. A herança em programação orientada a objetos tem exatamente esse significado.




Em algumas aplicações, as classes utilizadas possuem certos métodos e atributos em comum ou assinaturas iguais para métodos. Com uma linguagem de programação orientada a objetos nós podemos definir uma classe com diferentes níveis de abstração, permitindo decompor certos atributos comuns em diversas classes.

Uma classe genérica define então um conjunto de atributos que são compartilhados por outras classes.

Nós diremos que essas outras classes herdam da classe geral. A classe mais genérica é chamada de **superclasse** e as classes que herdam suas características são chamadas de **subclasses**.

02

Um aspecto muito importante da linguagem java é que uma classe pode herdar apenas de outra classe, isto é, **não existe o que chamamos de herança múltipla**.

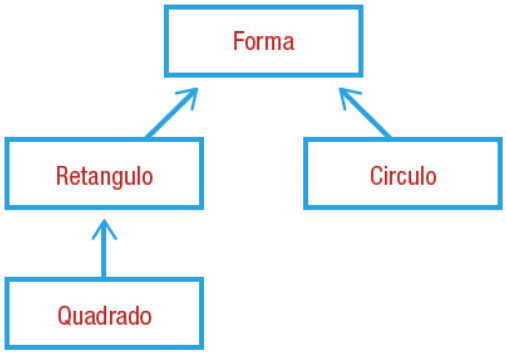
Uma classe pode herdar os métodos e atributos de outra classe. A herança é uma característica fundamental da programação orientada a objetos, pois garante a consistência entre os objetos e garante uma forma simples de criar novas classes.

Por exemplo, as classes Quadrado e Retangulo podem compartilhar o método area() que retorna o resultado do cálculo da área da figura. Melhor que escrever duas vezes o mesmo método, poderemos definir uma relação de herança entre as classes Quadrado e Retangulo. Nesse caso, somente a classe Retangulo contém o método área(), mas a classe Quadrado poderá utilizar esse método, caso a classe quadrado herde da classe Retangulo:


03

## 2 - Princípio da Herança

A ideia principal da herança é organizar as classes de maneira **hierárquica**. A relação de herança é unidirecional e, se uma classe B herda de uma classe A, nós diremos que B é uma subclasse de A. Essa noção de subclasse significa que a classe B é um caso particular, específico, da classe A e que os objetos que instanciam a classe B instanciam igualmente a classe A. Seja, por exemplo, as classes quadrado, retângulo e circulo. A figura a seguir propõe uma organização hierárquica dessas classes tal que Quadrado herda de Retângulo que herda, assim como a classe Círculo da classe Forma.




No momento, consideraremos a classe Forma vazia (sem nenhum atributo ou método) e nos focaremos mais especificamente às classes Retangulo e Quadrado.

04

A classe Retangulo herda de uma classe vazia, sendo assim ela não aproveita nenhum de seus atributos e deve definir todas as variáveis e métodos. Uma relação de herança é definida em Java por meio da palavra chave ***extends*** utilizada como no exemplo seguinte:

```
public class Retangulo extends Forma {
```

```
private int largura ;
private int comprimento ;
public Retangulo(int x, int y) {
this.largura = x ;
this.comprimento = y ;
}
public int getLargura() {
return this.largura ;
}
public int getComprimento() {
return this.comprimento ;
}
public int area() {
return this.comprimento * this.largura;
}
public void mostrar() {
   System.out.println(”retangulo ” + comprimento + ”x” + largura);
}
}
```

Em revanche, a classe Quadrado pode se beneficiar da classe Retangulo e não precisa reescrever os métodos que se aplicam à subclasse. Todos os métodos e variáveis da classe Retangulo são acessíveis dentro da classe Quadrado. Para que um atributo possa ser utilizado pela subclasse é preciso que seu tipo de acesso seja ***public*** ou ***protected***, ou se as duas classes pertencerem a um mesmo package que utilize um tipo de acesso padrão. No exemplo anterior, as variáveis comprimento e largura não são acessíveis dentro da classe quadrado, entretanto a classe poderá utilizar os métodos *getLargura*() e *getComprimento*() para obter os valores, visto que esses métodos foram declarados como ***public***.

Atenção: os atributos e métodos com o modificador ***private*** não são herdados!

05

## 3 - Redefinição de Métodos

A herança integral de atributos da classe Retangulo apresenta dois problemas:

1 - É necessário que cada quadrado possua largura e comprimentos iguais.
2 - O método mostrar escreve a palavra “Retangulo” no início do texto. Seria desejável que fosse escrito “Quadrado”.

Ainda, os construtores não são herdados por uma subclasse. É necessário então ter um construtor específico para a classe Quadrado. Isso nos permitiria resolver o primeiro problema, pois poderíamos ter um construtor que recebe apenas uma medida. Para atribuir esse valor aos atributos comprimento e largura (que são privados) deveremos lançar mão do construtor da classe Retangulo utilizando a palavra-chave **super**que chama o construtor da classe superior, como segue:

```
public Quadrado(int lado) {
super(lado, lado);
}
```


- A chamada a um construtor da classe superior deverá sempre estar localizada no construtor da classe e sempre como a primeira instrução
- Se nenhuma chamada ao construtor de uma classe superior for feita, o construtor chamará implicitamente um construtor vazio da classe superior (como se escrevêssemos *super*()). Se nenhum construtor vazio for acessível na classe superior, será gerado um erro na hora da compilação.

**06**

O segundo erro pode ser resolvido por uma redefinição de método. Nós dizemos que um método da subclasse redefine um método da sua classe superior, se eles têm a mesma assinatura, mas o tratamento efetuado é reescrito dentro da subclasse. Apresentamos a seguir o código da classe Quadrado no qual os dois problemas foram resolvidos.

```
public class Quadrado extends Retangulo {
public Quadrado(int lado) {
super(lado, lado);
}
public void mostrar() {
System.out.println(”Quadrado ” + this.getComprimento());
}
}
```

No momento da redefinição de um método, ou também chamada de sobrecarga, é ainda possível acessar o método que foi redefinido na classe superior. Esse acesso utiliza igualmente a palavra-chave **super** como prefixo para o método. No nosso caso, será necessário escrever*super.mostrar*() para efetuar o tratamento do método mostrar da classe Retângulo.

Por fim, é ainda possível impedir a redefinição de um método ou atributo incluindo a palavra-chave ***final*** no início da assinatura do método ou da declaração de um atributo. É também possível impedir a herança de uma classe utilizando também a palavra final no início da declaração da classe (antes da palavra-chave **class**). É importante salientar que os atributos definidos com o modificador final devem necessariamente ser inicializados na declaração ou pelo construtor da classe. Caso essa condição não seja satisfeita serão geradas advertências durante a compilação e será gerado um erro durante o processo de execução.

**07**

Vejamos a seguir mais um exemplo de herança entre classes. Temos uma classe pessoa que tem os dados básicos de cadastro de uma pessoa:


A classe **pessoa** é uma classe que permite manipular os dados básicos de uma pessoa. Mas suponha que queiramos manipular as informações de um funcionário. Obviamente, o funcionário é uma pessoa, logo a nova classe deve ter todas as características da classe pessoa, mas deve ter também outras informações específicas para o cadastro de um funcionário. Sendo assim podemos criar uma classe funcionário que herde as características da classe pessoa:


Então faríamos a declaração da classe funcionário da seguinte forma:

```
class funcionario extends pessoa{
}
```

A palavra ***extends*** assinala que a classe funcionário herda as características da classe pessoa.

08

Abaixo, apresentamos a implementação da classe pessoa.

**Exemplo _3_1_1: definição da classe pessoa**

```
import java.util.Calendar;

public class funcionario extends pessoa{

   public Calendar dataDeAdmissao;
   public String data;

   public funcionario(String Nome, String Tratamento, char sexo) {
       //O comando abaixo chama o construtor da classe pai
      super(Nome, Tratamento, sexo);
```

```
   }
   public funcionario(String Nome, String Tratamento, char sexo, String
dataDeAdmissao) {
      super(Nome, Tratamento, sexo);
      data = dataDeAdmissao;
      this.dataDeAdmissao = Calendar.getInstance();
      this.dataDeAdmissao = Calendar.getInstance();

this.dataDeAdmissao.set(Calendar.YEAR,Integer.parseInt(dataDeAdmissao.substring(6,
10)));
      this.dataDeAdmissao.set(Calendar.MONTH,
Integer.parseInt(dataDeAdmissao.substring(3, 5)));
      this.dataDeAdmissao.set(Calendar.DAY_OF_MONTH,
Integer.parseInt(dataDeAdmissao.substring(0, 2)));
  }

   public static void main(String[] args) {
      funcionario  func = new funcionario("Andrei","Sr.",'M',"02/01/2015");
      System.out.print(func.data.substring(0, 2)+func.data.substring(3,
5)+func.data.substring(6, 10));
        }
}
```

Vejam que não houve nenhuma diferença na declaração da classe pai. A declaração da classe **funcionario** é mostrada abaixo:

**Exemplo _3_1_2: definição da classe funcionário**.

Como vimos, o método **super** (linhas 10 e 14) chama o construtor da classe pai. No exemplo declaramos um objeto do tipo funcionario (linha 24). Entretanto, em função da herança poderíamos ter escrito:

```
funcionario  func = new funcionario("Andrei","Sr.",'M',"02/01/2015");
```

Ou

```
pessoa  func = new funcionario("Andrei","Sr.",'M',"02/01/2015");
```

Ou seja, um objeto da classe pai pode receber um objeto da classe filha. Uma forma simples de entender isso é que um funcionário é sempre uma pessoa. O contrário não é verdadeiro, ou seja, um objeto da classe filha não pode receber a referência de um objeto da classe pai. O código abaixo está errado:

```
//ATENÇÃO: CÓDIGO INCORRETO!!!
Funcionário func = new pessoa("Andrei","Sr.",'M',"02/01/2015");
```

09

## 4 - A CLASSE OBJECT

Como nós dissemos anteriormente, todas as classes derivam da superclasse ***Object***, portanto iremos descrever com mais detalhes essa classe e entender seus principais métodos. Isso é importante, porque todas as classes herdarão esses métodos que podem ser muito úteis no desenvolvimento da programação OO. A classe ***Object*** é declarada como segue:

```
public  class Object {
     public Object() {...}  // contrutor

     public String toString() {...}

     protected  native Object clone()  throws CloneNotSupportedException {...}

     public equals(java.lang.Object) {...}
     public  native  int hashCode() {...}

     protected  void finalize()  throws Throwable {...}

     public  final  native Class getClass() {...}

     // métodos utilizados na gestão de threads
     public  final  native  void notify() {...}
     public  final  native  void notifyAll() {...}

     public  final  void wait(long)  throws InterruptedException {...}
     public  final  void wait(long,  int)  throws InterruptedException {...}
}
```

10

**a) Método toString()**

Esse método é sem dúvida um dos mais utilizados da classe Object. Ele é utilizado sempre que temos necessidade de representar um objeto na forma de uma cadeia de caracteres. Por exemplo, o trecho de código abaixo é perfeitamente válido:

```
Retangulo ret = new Retangulo(5);
System.out.priuntln(“ret:”+ ret.toString() );
```

Como todas as classes derivam da classe Object, então a execução do código acima será feita por meio da chamada do método toString() da classe Object. Obviamente, como vocês devem imaginar se executarmos esse código irá aparecer na tela “ret:” seguido de uma cadeia de caracteres cabalísticos. Na verdade, a implementação padrão da classe toString() retorna uma cadeia de caracteres formada das seguintes partes:

- nome da classe
- caractere @
- endereço hexadecimal do endereço de memória no qual o objeto está armazenado.

Como podemos ver nada muito útil para o programador. Felizmente podemos redefinir o método toString() da classe Retangulo de forma a apresentar uma String em um formato pré-definido, por exemplo:

```
public class Retangulo  extends Forma {
        ...
        public String toString(){
                String resultado="";
                Resultado = "Retangulo - " + comprimento+" x "+largura;
                return Resultado;
}
}
```

11

**b) Método clone()**

O método *clone*() é um método do tipo ***native***, isso significa que ele não foi escrito em java, mas em alguma outra linguagem de programação como C ou C++. Java possui um mecanismo especial para passar parâmetros para métodos nativos, chamá-los e obter o retorno. Quem quiser mais detalhes sobre o assunto poderá consultar a API Java.

A função do método clone é duplicar rapidamente um objeto, duplicando a zona de memória na qual ele se encontra. Para duplicar um objeto basta chamar esse método que nos retornará uma cópia do objeto em questão.

Entretanto, é importante salientar que, por padrão, o Java proíbe a clonagem de objetos. Para que possa então duplicar um objeto você deverá sobrecarregar esse método que é ***protected*** na classe ***Object*** por um método ***public*** na classe do objeto que se quiser clonar. Além disso, a classe deverá implementar a interface ***Cloneable***. Essa interface não define nenhum método, mas serve para autorizar a execução da clonagem de uma instância.

Apresentamos a seguir um exemplo de sobrecarga na classe Retangulo:

```
public  class Retangulo extends Forma
 implements Cloneable {
        public Object clone()  throws CloneNotSupportedException {
        return  super.clone() ;
   }
}
```

12

**c) Método equals()**

Este método permite a comparação de dois objetos e, principalmente, se eles são iguais. Por padrão, esse método compara os endereços de memória dos dois objetos, caso sejam iguais ele retorna true, caso contrário, retorna false. Novamente para que esse método seja realmente útil deveremos redefini-

lo na nossa classe. O exemplo abaixo apresenta a implementação do método equals() para a classe Retangulo:

```
public  boolean equals(Object o) {
     if (!(o  instanceof Retangulo))
         return false ;

    Retangulo retangulo = (Retangulo) o ;

     return comprimento.equals(retangulo.comprimento) &&
           largura.equals(retangulo.largura)
}
```

Note que a primeira coisa que fizemos é usar o comando *instanceof* Retangulo para verificar se argumento é instância da classe Retangulo. Isso também nos garante que o argumento é não nulo. Uma vez que temos certeza que o objeto recebido é instância da classe Retangulo, criamos um objeto do tipo Retangulo para referenciá-lo e podermos acessar seus atributos. Por fim, definimos que consideraremos dois objetos da classe Retangulo iguais se os dois tiverem o mesmo comprimento e a mesma largura. Para isso, pode notar que usamos o método *equals*() também, mas nesse caso será chamado o método *equals*() da classe int.

13

**d) Método *hashCode*()**

A função do método *hashCode*() é calcular um valor de um código numérico para o objeto em questão.

Esse código numérico seria representativo desse objeto. Note que ele também é um método nativo. Por padrão o método retorna o endereço de memória do objeto, pois dois objetos não podem ocupar o mesmo endereço de memória, pelo menos não segundo as especificações da JVM da Sun. Entretanto, a especificação JAVA define que se dois objetos são iguais eles devem retornar o mesmo *hashCode*().

Essa condição acaba por relacionar o método equals() com o método *hashCode*(), de forma que, rigorosamente falando, ao redefinirmos o método equals() deveremos necessariamente redefinir também o método *hashCode*() de forma que esses objetos iguais tenham o mesmo*hashCode*.

14

**e) Método finalize()**

A apresentação do método finalize() é uma boa ocasião para falarmos sobre destruição de objetos.

Nós vimos como criar objetos e falamos, rapidamente, no início da disciplina, que uma das vantagens do Java é que ele possui um mecanismo chamado *Garbage Collector*, ou em tradução literal, coletor de lixo. Esse mecanismo funciona da seguinte forma: quando o *Garbage Collector*identifica a última referência aquele objeto, ele se encarrega de apagar o objeto da memória. Nesse momento o *Garbage Collector* chama esse método.

É importante salientar que apesar dos avanços das máquinas virtuais java ainda existem casos em que objetos não mais usados permanecem ocupando memória indefinidamente, sem serem percebidos pelo *Garbage Collector*.

15

**f) Método getClass()**

Esse método retorna um objeto que pe uma instância da classe Class. Algo interessante em java é que tudo é objeto em Java, inclusive as classes! Existe então uma classe chamada Class que serve para modelar as classes Java. O exemplo abaixo mostra como utilizar esse método para obter o nome da classe.

```
Retangulo m =  new Retangulo(5) ;
System.out.println("Classe Retangulo : " + m.getClass()) ;

> Classe Retangulo : class Retangulo
```

Para obter somente o nome da classe poderemos usar o método getName():

```
Retangulo m =  new Retangulo(5) ;
System.out.println("Classe Retangulo : " + m.getName()) ;

> Classe Retangulo : Retangulo
```

16

## RESUMO

Nesse módulo vimos o conceito de herança. Em certas aplicações, as classes utilizadas possuem em certos métodos e atributos em comum ou assinaturas iguais para métodos. Com uma linguagem de programação orientada a objetos nós podemos definir uma classe com diferentes níveis de abstração permitindo decompor certos atributos comuns em diversas classes. Uma classe genérica define então um conjunto de atributos que são compartilhados por outras classes. Nós diremos que essas outras classes herdam da classe geral.

Uma classe pode herdar os métodos e atributos de outra classe. A herança é uma característica fundamental da programação orientada a objetos, pois garante a consistência entre os objetos e garante uma forma simples de criar novas classes. A classe mais genérica é chamada de superclasse e as classes que herdam suas características são chamadas de subclasses.

Vimos ainda que podemos usar o mecanismo de redefinição de métodos, também chamado de sobrecarga, para escrever métodos em subclasses que substituam os métodos herdados da superclasse. A redefinição de métodos é fundamental para que as classes que herdem de uma classe comum possuam comportamentos específicos.

Por fim, apresentamos a classe Object que é a superclasse da qual todos os objetos são herdados. Descrevemos em detalhes seus métodos e apresentamos exemplos de como utilizá-los nas nossas classes.

## UNIDADE 3 – HERANÇA, POLIMORFISMO E CLASSES ABSTRATAS

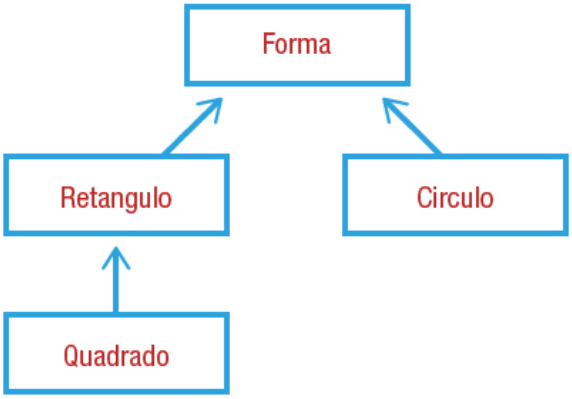
## MÓDULO 2 – POLIMORFISMO

01

### 1 - POLIMORFISMO: O QUE É

O polimorfismo é a capacidade atribuída a um objeto de ser uma instância de várias classes.

Vejamos o exemplo seguinte no qual temos a superclasse Forma, que possui outras três classes derivadas: Retangulo, Circulo e Quadrado:




Nesse caso existe uma única classe "real", que é aquela do construtor que foi chamado em primeiro lugar, exatamente o construtor que é chamado quando o objeto foi criado. Poderíamos, por exemplo, fazer a seguinte atribuição:

```
Forma form1;

Retangulo ret = new Retangulo();

form1 = ret;
```

02

Também podemos declarar o objeto com uma classe superior a sua classe real. Essa propriedade é muito útil para a criação de conjuntos que agrupam objetos de classes diferentes.

O exemplo seguinte armazena objetos de diferentes classes em um array do tipo da superclasse Forma.

```
Forma[] tabela = new Forma[4];
tabela[0] = new Retangulo(10,20);
```

```
tabela [1] = new Circulo(15);
tabela [2] = new Retangulo (5,30);
tabela [3] = new Quadrado(10);
```

O operador instanceof pode ser utilizado para testar a classe real de um objeto, conforme mostrado a seguir:

```
for (int i = 0 ; i < tabela.length ; i++) {
if (tabela[i] instanceof Forma)
System.out.println(”element ” + i + ” é uma forma”);
if (tabela [i] instanceof Circulo)
System.out.println(”element ” + i + ” é um círculo”);
if (tabela [i] instanceof Retangulo)
System.out.println(”element ” + i + ” é um retângulo”);
if (tabela [i] instanceof Quadrado)
System.out.println(”element ” + i + ” é um quadrado”);
}
```

A execução desse código para a tabela anteriormente definida mostrará o seguinte resultado:

```
element[0] é uma forma
element[0] é um retângulo
element[1] é uma forma
element[1] é um círculo
element[2] é uma forma
element[2] é um retângulo
element[3] é uma forma
element[3] é um retângulo
element[3] é um quadrado
```

03

O conjunto de classes Java, incluindo aquelas que não pertencem à API forma uma hierarquia com uma raiz única. Essa raiz é a classe ***Object***da qual todas as outras classes são herdadas. Na verdade, se não for mencionada explicitamente nenhuma relação de herança no momento de escrita da classe, por padrão, será considerada a herança da classe ***Object***. Graças a essa propriedade, as classes genéricas que agrupam conjuntos (Array, List etc.), agrupam objetos que pertencem à classe ***Object***.

Uma das propriedades que derivam do polimorfismo é que o interpretador Java é capaz de encontrar o tratamento adequado a ser utilizado quando chamamos um método de um objeto. Assim, mesmo com muitos objetos declarados na mesma classe, mas com diferentes classes reais, o interpretador será capaz de utilizar o método adequado à classe real. Se um método for redefinido para a classe real de um objeto então no momento da chamada será chamado o método da classe mais específica do objeto.

No exemplo seguinte, o método mostrar() foi redefinido em cada uma das classes para mostrar a informação correta do objeto específico:

```
for (int i = 0 ; i < tabela.length ; i++) {
tabela[i].mostrar();
```

```
}
```

O resultado da execução desse trecho de código seria:

```
retangulo 10x20
circulo 15
retangulo 5x30
quadrado 10
```

No estado atual das nossas classes, o código não poderá ser compilado. Na verdade a função mostrar() é chamada para cada um dos objetos mesmo que a classe forma não esteja definida na classe forma.

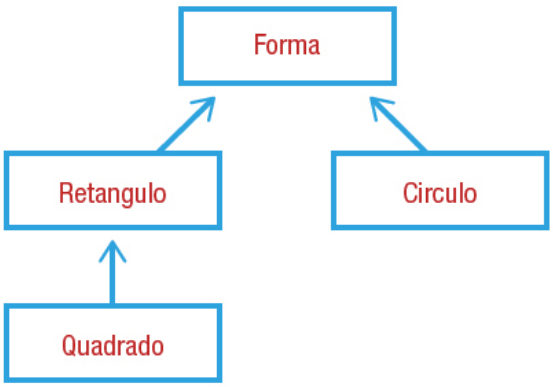
04

## 2 - POLIMORFISMO DE INCLUSÃO – SOBRESCRITA DE MÉTODOS

A sobrescrita de métodos pode ser classificada como um polimorfismo de inclusão.

A sobrescrita consiste na declaração de um método que sobrescreva o método herdado de uma classe.

Para que ocorra a sobrescrita dos métodos, ambos deverão ter a mesma assinatura, ou seja, o método da classe filha deverá ter o mesmo nome, mesmo valor de retorno e parâmetro. Vamos voltar ao exemplo das formas visto no módulo 1.




05

No exemplo tínhamos classes que abrangiam um método chamado mostrar(), o qual mostrava no console um texto com a descrição da forma.

Para que esse método mostrasse as informações corretas, tivemos que sobrescrevê-lo na classe quadrado, conforme abaixo:

```
public class Retangulo extends Forma {
…
public void mostrar() {
```

```
   System.out.println(”retangulo ” + comprimento + ”x” + largura);
}
}

public class Quadrado extends Retangulo {
...
public void mostrar() {
System.out.println(”Quadrado ” + this.getComprimento());
}
}
```

Podemos verificar que as assinaturas do método são exatamente iguais.

06

## 3 - POLIMORFISMO DE SOBRECARGA – SOBRECARGA DE MÉTODOS

Na sobrecarga de métodos, o método da classe filha possui uma assinatura ligeiramente diferente do método herdado. Embora os métodos tenham necessariamente de ter o mesmo nome, eles podem diferir pelos parâmetros do método e, facultativamente, pelo tipo de valor de retorno.


A linguagem java não aceita a sobrecarga de métodos que difiram apenas pelo tipo de valor de retorno.

O que é relevante enfatizar é que esses métodos sobrecarregados são novos métodos e não tem qualquer relação com o método herdado, a não ser a funcionalidade implementada.

Por exemplo, vamos sobrecarregar a função mostrar da classe Quadrado para que também possamos em lugar de mostrar no console retornar a String com o nome da forma e o comprimento:

```
public class Quadrado extends Retangulo {
...
public void mostrar() {
System.out.println(”Quadrado ” + this.getComprimento());
}
public String mostrar(int formato) {
if(formato==1){
return ”Quadrado”;
}
else{
return ”Quadrado ” + this.getComprimento().toString();
}
```

```
}
}
```

No exemplo acima, criamos um método mostrar, que recebe um parâmetro inteiro e retorna uma String. Dependendo do valor do formato, ele imprimirá apenas o nome da forma (“Quadrado”) ou o nome da forma seguido do comprimento.

**07**

A sobrecarga de métodos é muito usada nos construtores, o que fornece uma grande flexibilidade na criação de objetos permitindo a inicialização facultativa de atributos. Novamente iremos nos utilizar da classe pessoa descrita no módulo 1, que originalmente foi codificada como segue:

```
public class pessoa{

        public String Nome;
        public String Tratamento;
        public char sexo;

        public pessoa (String Nome, String Tratamento, char sexo){
                this.Nome=Nome;
                this.Tratamento = Tratamento;
                this.sexo = sexo;
        }
}
```

Podemos verificar que o construtor é bastante rígido, visto que obriga a chamada com a especificação dos três atributos da classe.

**08**

Uma forma de facilitar a criação de objetos do tipo pessoa seria sobrecarregar esse construtor, por exemplo:

```
public class pessoa{

        public String Nome;
        public String Tratamento;
        public char sexo;

        public pessoa (String Nome){
                this.Nome=Nome;
                this.Tratamento = “”;
                this.sexo = “”;
        }

        public pessoa (String Nome, String Tratamento){
                this.Nome=Nome;
                this.Tratamento = Tratamento;
                this.sexo = “”;
```

```
        }

        public pessoa (String Nome, String Tratamento, char sexo){
                this.Nome=Nome;
                this.Tratamento = Tratamento;
                this.sexo = sexo;
        }
}
```

Então na declaração de um objeto do tipo pessoa, poderíamos optar pelo construtor mais adequado:

```
pessoa pepe = new pessoa(“pepe”);
```

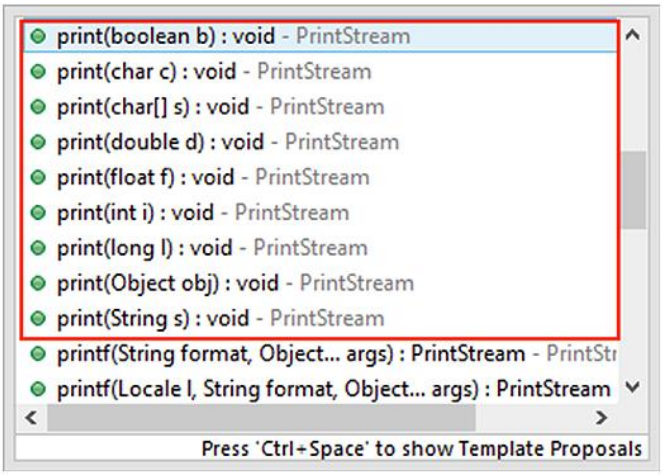
Ou

```
pessoa pepe = new pessoa(“pepe”,”Sr.”);
```

09

Exemplos de polimorfismo de sobrecarga são muito comuns em métodos da biblioteca java.

O método System.out.print() apresenta diversas sobrecargas, como podemos ver nas sugestões de preenchimento do Eclipse:




Desde que a sobrecarga seja feita de forma cuidadosa, mantendo a funcionalidade básica do método, ela é uma ótima ferramenta para aumentar a flexibilidade e a usabilidade do código.

10

## Resumo

O polimorfismo é um dos pilares da programação orientada a objetos. O polimorfismo que significa “diferentes formas” consiste na capacidade atribuída a um objeto de ser uma instância de várias classes.

No caso um objeto da classe pai pode ser instância de objeto de classes descendentes. Essa capacidade fornece ao programador maior flexibilidade, pois o tipo de objeto instanciado poderá ser definido apenas em tempo de execução. Isso permite, por exemplo, criar arrays que armazenem objetos de diferentes subclasses.

Vimos ainda nesse módulo o polimorfismo de inclusão que se materializa no código java por meio da sobrescrita de métodos herdados. Nesse caso, um método da subclasse terá exatamente a mesma assinatura do método herdado, porém com uma implementação diferente. No polimorfismo de sobrecarga, a subclasse poderá sobrecarregar métodos herdados. Os métodos sobrecarregados deverão necessariamente ter o mesmo nome do método herdado, mas obrigatoriamente com parâmetros diferentes. O tipo de valor de retorno poderá ou não ser diferente. Vimos que a sobrecarga de métodos é uma forma simples de tornar a classe mais flexível, bem como melhorar a usabilidade do código.

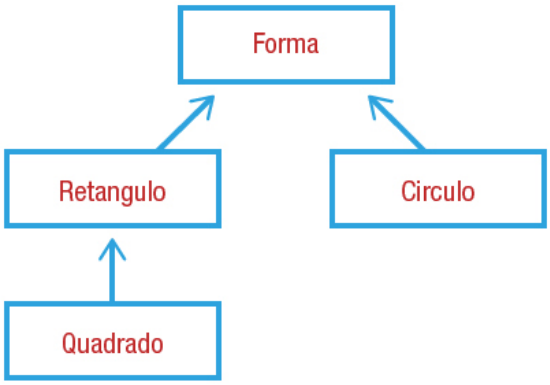
## UNIDADE 3 – HERANÇA, POLIMORFISMO E CLASSES ABSTRATAS

## MÓDULO 3 – TÉCNICAS DE ABSTRAÇÃO

01

## 1 - INTERFACES

Vale lembrar que a criação de classes por meio de herança é muito útil quando as subclasses compartilham a implementação de métodos e atributos da classe pai. Conforme já vimos, podíamos criar uma classe Forma e derivar diversas subclasses como Retângulo, Quadrado e Círculo.




Entretanto, se prestarmos mais atenção ao código que escrevemos, podemos verificar que a classe Forma não acrescenta muito nas demais classes, visto que existem diversas formas diferentes (oval, circulo, retângulo, quadrado etc.) e cada uma tem métodos muito particulares e atributos diferentes. Assim pudemos perceber que não se ganhou muito com a herança da classe Forma, pois não houve praticamente nenhum reaproveitamento de atributos e da implementação de métodos. O que fizemos foi usar apenas as assinaturas dos métodos da classe forma e tivemos que sobrescrevê-los.

Se não houver um real ganho de implementação de código, não é recomendado o uso de herança de classes reais, e sim o uso de interfaces e, em um segundo momento, de classes abstratas.

02

A ideia que está por trás do conceito de interface é de garantirmos que as classes possuem determinados métodos. Podemos pensar a interface como um carimbo de garantia de que a classe, ou um conjunto de classes, possui a implementação de determinados métodos. Uma **interface**é definida pela palavra-chave interface. Mostramos a seguir a implementação da interface Forma:

```
public interface Forma {
        public float area();
        public float perimetro();
        public void mostrar();
}
```

A interface Forma compreende três métodos:

- área,
- perímetro e
- mostrar.

Vejam que a interface não define atributos, contém apenas as assinaturas dos métodos. Vejam que a interface Forma define que as classes que implementarem forma devem, necessariamente:

- implementar um método chamado area que não receberá nenhum argumento, mas retornará um valor do tipo float que corresponderá a área do objeto.
- implementar um método chamado perimetro que não receberá nenhum argumento, mas retornará um valor do tipo float que corresponderá ao perímetro do objeto.
- implementar o método mostrar que informa o nome da figura e suas dimensões principais.

03

Agora vamos alterar as classes Retangulo, Quadrado e Circulo para que implementem a interface forma (e deixe de herdar da classe forma!). Para isso, iremos usar a palavra-chave ***implements***.

```
public class Retangulo implements Forma{
        float largura;
        float comprimento;

        public Retangulo(float largura, float comprimento){
                this.largura=largura;
                this.comprimento=comprimento;
        }
        public float area(){
                return largura*comprimento;
        }
```

```
        public float perimetro(){
                return 2*largura+2*comprimento;
        }
        public void mostrar(){
                System.out.printf("Retangulo %f x %f",comprimento, largura);
        }
}
```

```
public class Quadrado extends Retangulo implements Forma{
        public Quadrado(float largura){
                super(largura,largura);
        }

        public void mostrar(){
                System.out.printf("Quadrado com %f de lado \r",largura);
        }
}
```

```
public class Circulo implements Forma{
        float raio;

        public Circulo(float raio){
                this.raio=raio;
        }

        public float area(){
                return (float)Math.PI*raio*raio;
        }
        public float perimetro(){
                return (float)Math.PI*raio*2;
        }
        public void mostrar(){
                System.out.printf("Circulo de  %f de raio",raio);
        }
}
```

04

Um ponto fundamental do uso de interfaces é saber que uma interface não pode ser instanciada, lembre-se que se trata apenas de um “contrato” de requisitos para a classe. Sendo assim, o trecho de código abaixo não é correto:

```
Forma figura = new Forma();     // INCORRETO!!!
```

Entretanto, você pode utilizar a interface de um jeito muito interessante, pois ela pode instanciar objetos que implementem a interface. Assim sendo, o código abaixo é perfeitamente correto:

```
public class Principal {
        public static void main(String[] args) {
```

```
                Forma figura1 = new Retangulo(5,4);
                Forma figura2 = new Circulo(4);
                Forma figura3 = new Quadrado(4);
                figura1.mostrar();
                figura2.mostrar();
                figura3.mostrar();
        }
}
```

**05**

Muitas vezes, ao aprendermos interfaces, achamos que são códigos que não servem para nada, já que não é feita a implementação propriamente dita dos métodos. Entretanto a interface nos possibilita um nível de abstração maior, tornando o código muito mais flexível, pois podemos mudar a implementação sem maiores impactos no restante do código, desde que mantenhamos a implementação das interfaces intactas.

O exemplo a seguir ilustra isso mostrando o uso de interfaces como parâmetros de um método:

```
public class Principal {

        public static void mostrar(Forma form){
                form.mostrar();
        }

        public static void main(String[] args) {
                Retangulo figura = new Retangulo(5,4);
                mostrar(figura);
        }
}
```

Perceba que o método mostrar da classe Principal recebe como argumento qualquer objeto que implemente a interface Forma, ou seja, não é necessário que o objeto seja herdado de uma classe específica e nem há qualquer restrição em relação a forma como os métodos são implementados. Obviamente, deve-se ter um cuidado para que o método ou função que tem interfaces como parâmetros utilizem apenas o que foi definido na interface em sua implementação.

**06**

Se uma classe implementar uma interface, mas o programador não escreveu a implementação de todos os métodos da interface será gerado um erro de compilação, a menos que a classe seja abstrata (assunto que veremos a seguir).

A partir da versão Java 8 é possível incluir nas interfaces um método concreto, ou seja, com a implementação padrão do método. Essa característica é muito útil e evita a implementação desnecessária. Para definir um método padrão usamos a palavra *default* antes do nome do método.


Alertamos apenas para o fato que as versões antigas de java não são compatíveis com a declaração default o que pode gerar erros e incompatibilidades.

Sugerimos que sempre que possível sejam usadas interfaces em lugar de herança, pois tornamos o código mais flexível.

Um exemplo interessante disso é o método *Collections.sort*(). Esse método permite ordenar qualquer lista de objetos desde que implementem a interface ***Comparable***.

07

## 2 - Classes Abstratas

O conceito de classe abstrata se situa entre o conceito de classe e o de interface. É uma classe que não se pode instanciar, pois determinados métodos não são implementados.

Uma classe abstrata pode conter atributos, métodos implementados e assinaturas de métodos a implementar. Uma classe pode implementar totalmente ou parcialmente uma interface e pode herdar de uma classe ou de uma classe abstrata.

**Qual a vantagem de se ter classe que não pode ser instanciada?**

Se retornarmos ao nosso exemplo, podemos ver que Forma é um conceito genérico. Não desejamos criar uma forma, e sim figuras que contenham características gerais de uma forma, mas com formatos específicos como quadrados, círculos e retângulos. Assim não teria sentido ter um objeto do tipo Forma, não é mesmo? Entretanto, algumas características de uma forma são compartilhadas por todas as figuras como origem, então temos aqui uma necessidade de herdar atributos, uma flexibilidade que a interface não possui.

A palavra abstract é utilizada antes da palavra-chave ***class*** para declarar uma classe abstrata, bem como para declarar as assinaturas de métodos a implementar.

08

Imagine que desejemos ter dois atributos: origem_x e origem_y, em todos os objetos que representem uma Forma. Como uma interface não pode conter atributos, será preciso converter Forma para uma classe abstrata que chamaremos de Forma2, como segue:

```
public abstract class Forma2 {
        //Atributos
```

```
        private int origem_x ;
        private int origem_y ;
        //Construtor
        public Forma2() {
                this. origem_x = 0;
                this. origem_y = 0;
        }
        //Classes reais
        public int getOrigemX() {
                return this.origem_x;
        }
        public int getOrigemY() {
                return this.origem_y;
        }
        public void setOrigemX(int x) {
                this.origem_x = x;
        }
        public void setOrigemY(int y) {
                this.origem_y = y;
        }
        //Classes abstratas
        public abstract float area();
        public abstract float perimetro();
        public abstract void mostrar();
}
```

Note que a declaração de métodos abstratos foi feita incluindo a palavra-chave abstract antes do nome do método e terminando a declaração por ponto e vírgula em lugar de abre e fecha chaves.

09

Assim que uma classe herda de uma classe abstrata, ela deve:

- implementar os métodos abstratos de sua superclasse, incluindo o conteúdo.
- ou ela mesma deverá ser uma classe abstrata, se pelo menos um dos métodos abstratos continuar sem ser implementado.

Ilustraremos a herança de uma classe abstrata reestabelecendo a herança das classes Retangulo e Circulo em relação à Forma2. Para facilitar nomearemos essas classes de Retangulo2 e Circulo2.

```
public class Retangulo2 extends Forma2{
        float largura;
        float comprimento;

        public Retangulo2(float largura, float comprimento){
                this.largura=largura;
                this.comprimento=comprimento;
        }

        public float area(){
                return largura*comprimento;
```

```
        }
        public float perimetro(){
                return 2*largura+2*comprimento;
        }
        public void mostrar(){
                System.out.printf("Retangulo %f x %f\r",comprimento, largura);
        }
}
```

e

```
public class Circulo2 extends Forma2{
        float raio;

        public Circulo2(float raio){
                this.raio=raio;
        }

        public float area(){
                return (float)Math.PI*raio*raio;
        }
        public float perimetro(){
                return (float)Math.PI*raio*2;
        }
        public void mostrar(){
                System.out.printf("Circulo de  %f de raio\r",raio);
        }
}
```

10

A flexibilidade de utilização das classes abstratas é similar às interfaces. Podemos utilizar as classes **abstract** para instanciar objetos de classes herdadas (Atenção: desde que sejam classes reais!):

```
public class Principal {

        public static void mostrar(Forma2 form){
                form.mostrar();
        }

        public static void main(String[] args) {
                Forma2 figura = new Retangulo2(5,4);
                mostrar(figura);
        }
}
```

O exemplo abaixo mostra a declaração de uma classe polígono que não possui a implementação dos métodos area() e perimetro(), portanto a classe teve ser que declarada como abstract:

```
public abstract class Poligono extends Forma2{
        int nlados;
```

```
        float lado;

        public Poligono(int nlados, float lado){
                this.nlados=nlados;
                this.lado=lado;
        }
        public void mostrar(){
                System.out.printf("Poligono com  %d lados de %f\r",nlados, lado);
        }
}
```

11

Como dissemos, essa classe obrigatoriamente manteve o modificador abstract porque não implementou todos os métodos abstract da superclasse. A consequência é que essa classe não poderá ser instanciada.

O código abaixo gerará um erro:

```
public class Principal {

        public static void mostrar(Forma2 form){
                form.mostrar();
        }

        public static void main(String[] args) {
                Forma2 figura = new Poligono(5,4);
                mostrar(figura);
        }
}
```

12

## 3 - Classes e métodos genéricos

Estudamos anteriormente as coleções e nos deparamos com classes ArrayList e LinkedList que recebiam um argumento especial que correspondia a um tipo de dado:

```
List<int> valores = new ArrayList<int>();
```

Nesse trecho de código estamos criando um objeto do tipo ArrayList para dados do tipo int. Algumas classes exigem esse tipo de parametrização, ou seja, a especificação do tipo de dado a ser usado pelo objeto.

A classe ArrayList é um exemplo dessa parametrização, mas no pacote java.util, existem muitas outras classes genéricas, principalmente, as classes que representam coleções que precisam saber o tipo da dado a ser processado (Vector, LinkedList etc.). Essas classes são genéricas no sentido que tomam como parâmetro um tipo de dado (classe ou interface) qualquer **E**. O **E** é semelhante a uma variável que recebe como valor o tipo de dado. Podemos ver abaixo um exemplo do pacote java.util.ArrayList:

```
package java.util ;
public class ArrayList<E> extends AbstractList<E>
implements List<E>, ...
{
...
public boolean add(E e) {
...
}
...
}
```

Nós podemos notar que o tipo passado como parâmetro é indicado entre os símbolos <> (Ex.: <E>) e pode ser reutilizado pela classe dentro de métodos (Exemplo: método set()). No caso anterior o método Add define como parâmetro um valor do tipo E. Por exemplo, a função Collections.sort(List<T>) ordena os elementos de uma lista.

13

O mais interessante aqui é que a classe é genérica, o que permite a ordenação de qualquer tipo de objeto. O objeto nem precisa ser numérico, basta que implementa a interface Comparable. Isso permitiria, por exemplo, ordenar uma lista de objetos do tipo funcionario usando esse método, como mostrado no exemplo a seguir:

```
public class funcionario implements Comparable<funcionario>{
        private String nome;
        private int matricula;

        public funcionario(String nome, int matricula){
                this.nome=nome;
                this.matricula=matricula;
        }
        public String getNome(){
                return nome;
        }
        public int getValue() {
                return this.matricula;
        }

        public int compareTo(funcionario o){
                if (this.matricula < o.matricula) {
                        return -1;
                }
                if (this.matricula == o.matricula) {
                        return 0;
                }
                assert this.matricula > o.matricula;
                        return 1;
                }
}
```

14

No exemplo anterior temos a implementação da interface Comparable na classe funcionário e então podemos seguir com a ordenação, conforme exemplo abaixo:

```
import java.util.ArrayList;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class principal {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		funcionario[] cadastro = new funcionario[5];

		List<funcionario> cadastroSorted = new ArrayList<funcionario>();
		cadastroSorted.add(new funcionario("Pedro",3));
		cadastroSorted.add(new funcionario("Joao",2));
		cadastroSorted.add(new funcionario("Maria",5));
		cadastroSorted.add(new funcionario("Roberta",1));
		cadastroSorted.add(new funcionario("Carlos",4));

		Collections.sort(cadastroSorted);

		for(funcionario func:cadastroSorted){
			System.out.println(func.getNome());
		}
	}
}
```

O exemplo acima mostra que é fácil utilizarmos as classes genéricas e nos poupa bastante trabalho, pois já existem implementações de funções de pesquisa e ordenação prontas e que podem ser utilizadas, bastando para isso implementarmos as interfaces necessárias.

15

## RESUMO

Nesse módulo vimos dois conceitos muito importantes: interface e classes abstract. Uma interface é um tipo, da mesma forma que uma classe, mas abstrata e não pode ser instanciada (chamando **new** mais o construtor). Uma interface descreve um conjunto de assinaturas de métodos, sem implementação, que devem ser implementados em todas as classes que implementam essa interface. É como se fosse um contrato ou uma especificação dos métodos mínimos que uma classe deve ter para receber um determinado status.

A utilidade do conceito de interface reside no reagrupamento de diversas classes, de forma que cada uma implementa um conjunto comum de métodos, sob um mesmo tipo. Uma interface possui as seguintes características:

- Contém assinaturas de métodos
- Não pode conter atributos

- Uma interface pode ser herdada de outra interface (com a palavra-chave **extends**).
- Uma classe pode implementar diversas interfaces. A lista de interfaces implementadas deve ser colocada após a palavra-chave**implements** colocada na declaração da classe, separando cada interface com uma vírgula.

As classes abstratas são classes que não possuem implementação para pelo menos um de seus métodos. Para declarar uma classe ou um método abstrato utilizamos a palavra-chave **abstract**. Uma classe abstrata não pode ser instanciada.

As classes filhas de uma classe abstrata herdarão todos os atributos e métodos implementados e poderão ou não implementar os métodos abstratos. Caso implementem todos os métodos abstratos então poderão ser instanciadas normalmente. Caso não implementem, então serão por sua vez também classes abstratas. Podemos considerar que as classes abstratas trazem a vantagem da herança de códigos de implementação, com a flexibilidade dos métodos abstratos que podem ou não ser implementados pelas demais classes filhas.

Por fim, apresentamos a descrição e uso de classes genéricas. As classes genéricas necessitam de um parâmetro adicional com o tipo de dado a ser processado pela classe. As classes genéricas mais usadas são aquelas que envolvem as coleções de dados e seus processos de pesquisa e ordenamento. Ilustramos o uso das classes genéricas com um exemplo completo de ordenamento de um array de funcionários pelo número da matrícula.

## UNIDADE 3 – HERANÇA, POLIMORFISMO E CLASSES ABSTRATAS
## MÓDULO 4 – CRIANDO INTERFACES GRÁFICAS PARA O USUÁRIO
**01**

# 1 - BIBLIOTECAS JAVA PARA CRIAÇÃO DE INTERFACE GRÁFICA

Neste módulo veremos como desenvolver rapidamente aplicações Java com interface gráfica. Há vasta quantidade de informações na Web a respeito deste assunto. Então, recomendamos que você pesquise, crie desafios e teste para estender os conhecimentos.

Existem várias bibliotecas para criação de interface gráfica em aplicações “Desktop” (que são executadas localmente em um computador) com Java. As principais são:

### 1.1 AWT

Foi o primeiro toolkit a ser criado no Java para implementação de interfaces gráficas. Possui um conjunto de classes para desenhos 2D e componentes de tela (classes *button, image, canvas, font, color* etc.).

Para mais informações acesse o site.


## Exemplo: Criando um formulário com AWT

Para isso, vamos criar um projeto **Ex1AWT** no ambiente Eclipse, seguindo o procedimento adotado nas unidades anteriores:

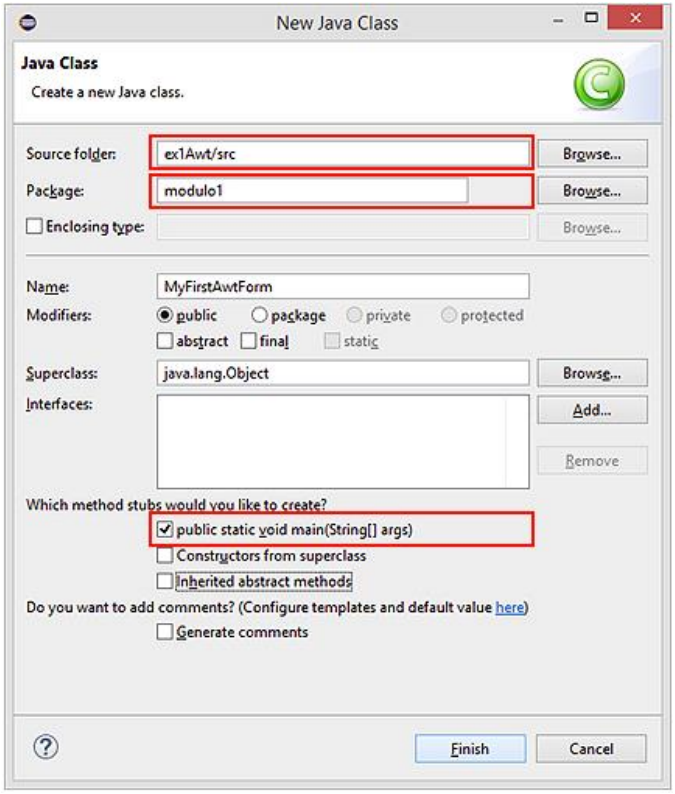
1. Vá em **File** – **New** – **Java Project**
2. Na tela **New Java Project**, em **Project Name**, digite **Ex1AWT**
3. Aperte **Finish**

Não é necessário adicionar nenhuma biblioteca adicional (jar). O AWT acompanha a maioria das distribuições Java.

02

Então, vamos criar uma classe **MyFirstAwtForm** que será nosso aplicativo, da seguinte forma:

1. Na aba **Package Explorer**, clique no projeto com o botão esquerdo e escolha as opções **New – Class**
2. Na tela **New Java Class**, em **Name**, coloque o nome de sua classe que implementará o app: **MyFirstAwtForm**
3. Escolha as opções conforme a imagem abaixo:




4. Aperte **Finish**

03

A classe gerada terá então o seguinte código:

```
package modulo1;

public class MyFirstAwtForm {

    /**
   * @param args
   */
    public static void main(String[] args) {
    }

}
```

Para utilizarmos as classes AWT precisamos “importar” as bibliotecas AWT adicionando o seguinte código após a linha com package:

```
import java.awt.*;
import java.awt.event.*;
```

Vamos então criar o formulário com a classe **Frame** do AWT. Para isso, digite o código abaixo na função **main**:

```
Frame frm = new Frame("Minha primeira tela");
```

Se rodarmos o programa agora nada acontecerá: o fluxo do programa irá direto ao fim do **main** e o programa terminará. Isto ocorre porque precisamos fornecer ao nosso programa um "tratador de eventos" para este frame. Isto é feito com interfaces **Listener**.

04

No caso de uma janela (frame), a interface Listener é implementada pela classe **WindowAdapter**. No caso de uma janela (frame), a interface Listener é implementada pela classe **WindowAdapter**. Então precisamos adicionar no código o trecho que define a classe FecharJanela conforme abaixo. Não se esqueça que a referida classe FecharJanela deve estar no mesmo arquivo e fora do escopo da classe MyFirstAwtForm.

```
class FecharJanela extends WindowAdapter{
@Override
public void windowClosing(WindowEvent e) {
System.exit(0);
}
}
```

```
frm.setSize(350, 200);
frm.setVisible(true);
frm.addWindowListener(new FecharJanela(){

    public void windowClosing(WindowEvent e){
        System.exit(0);
    }
});
```

- **setVisible** e **setSize** são métodos da classe Frame que ajustam a visibilidade e o tamanho da tela.
- **addWindowListener** também é um método da classe Frame, que indica qual será o "Listener" que irá tratar os eventos desta tela.

Entendemos como **eventos** tudo que pode ocorrer num elemento de interface gráfica ao longo do tempo.

Exemplos:

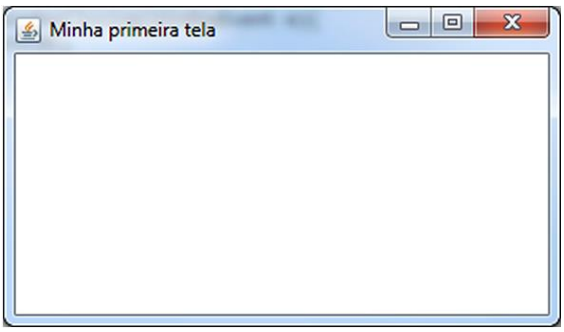
- Um clique no botão “fechar” da tela
- O mouse passando sobre a tela
- A tela (frame) perdendo o foco etc.

Para cada um destes eventos existe um método do Listener para tratá-lo.

05

No exemplo anterior, o evento tratado é o de “fechando a janela” (windowClosing).

Se rodarmos o programa agora, a seguinte tela será mostrada:

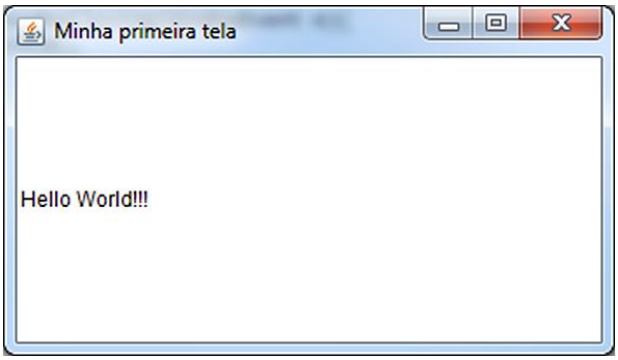



Note que, por ser AWT, o estilo da tela é a do sistema operacional que estamos utilizando, que neste exemplo é o Windows.

Para finalizar, colocaremos um **label** escrito “Hello World” em nosso formulário. Então, digite o seguinte código logo após criar o Frame:

```
Label lbl = new Label("Hello World!!!");
frm.add(lbl);
```

Agora, nosso aplicativo ficou muito mais amigável!




06

### 1.2 Swing

É um toolkit derivado do AWT, de mais alto nível, que também define um conjunto de classes para componentes de tela (classes *JButton, JFrame, JMenu* etc.)

A principal diferença com o AWT é que a Swing oferece mais independência do sistema operacional utilizado, sendo recomendada para desenvolvimento multiplataforma.

Isto significa que, se você fizer um programa de interface gráfica para um determinado sistema operacional, o comportamento da interface do programa será idêntico se você rodá-lo em outro sistema operacional.

Assim como o AWT, praticamente todas as distribuições Java incluem este toolkit, e não necessitam serem instalados separadamente.

Para mais informações acesse o site.

07


### Exemplo: criando um formulário Swing

Para verificarmos como funciona o Swing, da mesma forma que criamos o projeto **Ex1AWT**, vamos criar o **Ex3Swing**.

Agora, vamos criar a classe **MyFirstSwingForm** (da mesma forma que o MyFirstAwtForm) conforme a tela abaixo:

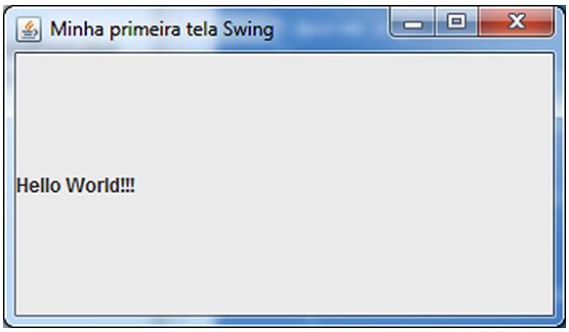

08

O código-fonte da classe **MyFirstSwingForm** será o abaixo:

```
package modulo1;

import java.awt.event.*;
import javax.swing.*;

public class MyFirstSwingForm {

    /**
   * @param args
   */
    public static void main(String[] args) {

        JFrame frm = new JFrame("Minha primeira tela Swing");

        frm.setVisible(true);
        JLabel lbl = new JLabel("Hello World!!!");
        frm.add(lbl);
        frm.setSize(350,200);

        frm.addWindowListener(new WindowAdapter()
```

```
        {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
    }
}
```

O que está diferente do AWT? Apenas o "J" na frente das classes Frame e Label?
A implementação interna das classes Swing são diferentes. Percebemos isto quando executamos o programa e notamos que a visualização do label é ligeiramente diferente (está com outra fonte):




09

### 1.3 SWT

É uma alternativa mais moderna às duas bibliotecas anteriores.

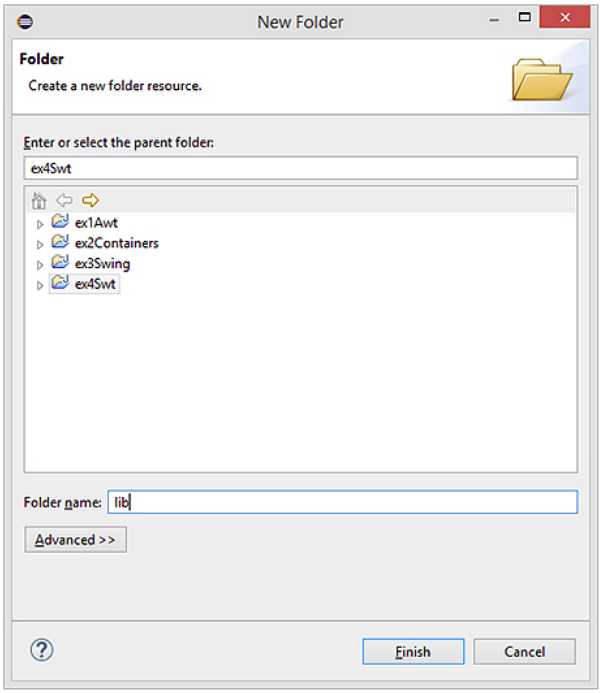
Utiliza bibliotecas nativas do sistema operacional para gerar a visualização de seus elementos gráficos. Por isso, apresenta uma performance melhor. É mantida pela mesma equipe do Eclipse.

Para mais informações e download acesse o site.


**Exemplo: criando um formulário SWT**

Diferentemente das bibliotecas anteriores, o SWT tem distribuição separada. Você precisa fazer *download* do pacote "jar". Para isto:

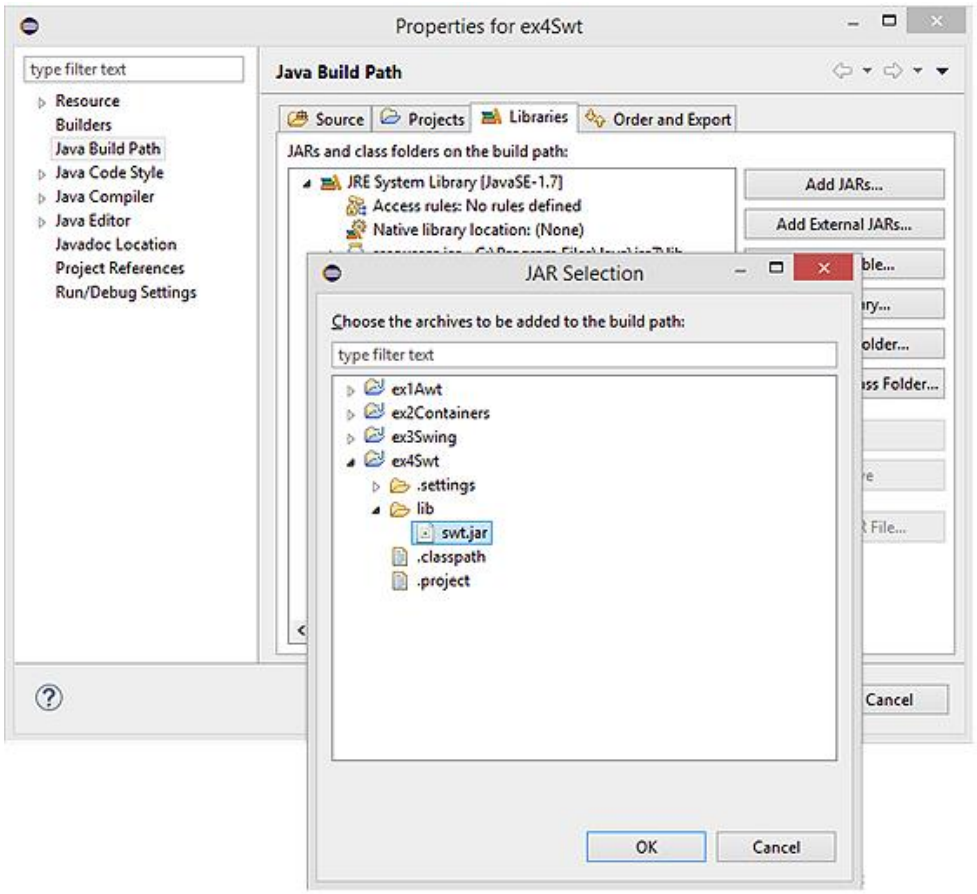
- Acesse https://www.eclipse.org/swt/ e
- baixe a última versão estável do swt.
- Descompacte o arquivo zip, e copie o arquivo **swt.jar** em uma pasta “lib” em seu projeto
- Para criar a pasta “lib”, clique com botão esquerdo no projeto e selecione as opções **New – Folder**. A seguinte tela é mostrada:




- Digite “lib” e aperte **Finish**.
- Para copiar o **swt.jar** nesta pasta basta dar “Ctrl+C” no explorer e “Ctrl+V” no eclipse, com o foco do Package Explorer sobre a pasta lib.

10

A seguir, para que o compilador java do eclipse saiba que a biblioteca swt existe, é necessário incluí-la no **build path**. Para isto, vá em**properties** do seu projeto e clique na aba **Libraries**, e a seguir, no botão **Add JARS**... conforme as telas abaixo:




Feito isto, crie uma classe **MyFirstSwtForm**, e copie o código abaixo:

```
package modulo1;

import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Point;
import org.eclipse.swt.graphics.Rectangle;
import org.eclipse.swt.layout.FormAttachment;
import org.eclipse.swt.layout.FormData;
import org.eclipse.swt.layout.FormLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Event;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Listener;
import org.eclipse.swt.widgets.Shell;

public class MyFirstSwtForm {

    /**
   * @param args
   */
    public static void main(String[] args) {
        Display display = new Display();
```

```
        final Shell shell = new Shell(display);

        Label label = new Label(shell, SWT.WRAP);
        label.setText("Alo! Mundo!");

        Button button1 = new Button(shell, SWT.PUSH);
        button1.setText("OK");
        Button button2 = new Button(shell, SWT.PUSH);
        button2.setText("Cancel");

        final int insetX = 4, insetY = 4;
        FormLayout formLayout = new FormLayout();
        formLayout.marginWidth = insetX;
        formLayout.marginHeight = insetY;
        shell.setLayout(formLayout);

        Point size = label.computeSize(SWT.DEFAULT, SWT.DEFAULT);
        final FormData labelData = new FormData(size.x, SWT.DEFAULT);
        labelData.left = new FormAttachment(0, 0);
        labelData.right = new FormAttachment(100, 0);
        label.setLayoutData(labelData);

        shell.addListener(SWT.Resize, new Listener() {
          public void handleEvent(Event e) {
            Rectangle rect = shell.getClientArea();
            labelData.width = rect.width - insetX * 1;
            shell.layout();
          }
        });

        FormData button2Data = new FormData();
        button2Data.top = new FormAttachment(label, insetY);
        button2Data.right = new FormAttachment(100, -insetX);
        button2Data.bottom = new FormAttachment(100, 0);
        button2.setLayoutData(button2Data);

        FormData button1Data = new FormData();
        button1Data.right = new FormAttachment(button2, -insetX);
        button1Data.bottom = new FormAttachment(100, 0);
        button1.setLayoutData(button1Data);

        shell.pack();
        shell.open();
        while (!shell.isDisposed()) {
          if (!display.readAndDispatch())
            display.sleep();
        }
        display.dispose();
    }
}
```

11

# 2 - USANDO O SWING

### 2.1 Classes, componentes e containers do Swing

Vamos estudar um pouco mais as classes Swing, que serão fundamentais em nossa disciplina.

Podemos agrupar os elementos de interface em 2 grupos:

1. **Containers**

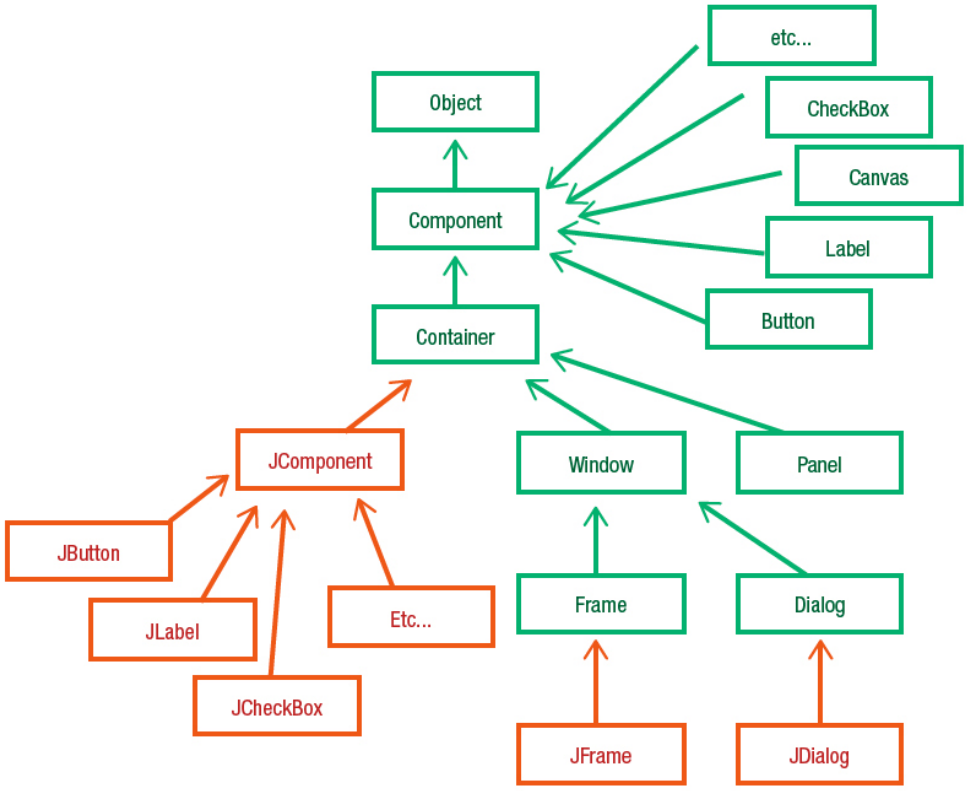
   Servem para agrupar e exibir outros componentes (JPanel, JFrame, JApplet...)

No exemplo **MyFirstSwingForm**, observamos que há um container JFrame que contém um componente JLabel.

Observando a hierarquia das classes Swing / AWT percebe-se que esta divisão existe bem definida no AWT, mas no Swing é um pouco diferente:

2. **Componentes**

   São componentes atômicos, ou seja, que não permitem outros componentes dentro deles (JButton, JLabel, JTextField, JScrollBar...)




12

**Exemplo: usando containers**

Para ilustrar o uso do container, crie um novo projeto chamado ex2Containers, crie uma classe com 'main' e adicione o código abaixo:

```
public class TesteContainers {

    /**
   * @param args
   */
    public static void main(String[] args) {

        JFrame frm = new JFrame("Teste de containers");

        // Ajusta posição e tamanho da janela principal
        frm.setBounds(25, 100, 400, 150);

        frm.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
```

```
        // Define o layout do container (frm)
        FlowLayout flow = new FlowLayout();

        // Define o tamanho
        Container container = frm.getContentPane();

        // Ajusta layout do container
        container.setLayout(flow);

        // Adiciona 4 botões
        for (int i=1; i<=4; i++)
            container.add(new JButton("Aperte " + i));

        // Exibe a janela
        frm.setVisible(true);
    }
}
```

Na instrução **container.add** observamos que um botão é adicionado no container **frm**. Da mesma forma poderíamos adicionar outros componentes. A instrução **frm.setDefaultCloseOperation** ilustra uma forma de implementarmos a tela sem criarmos um listener para tratar o evento “fechar tela”.

13

### 2.2 Layouts

No exemplo anterior, utilizamos uma classe **FlowLayout** que é utilizado para definir o posicionamento dos botões no container. Esta classe é um **LayoutManager**. Em Java utilizamos gerenciadores de layout, ao invés de posicionarmos os componentes através de coordenadas x e y, porque a ideia é que a interface seja extremamente portável e escalável. Os principais gerenciadores de layout são:

• FlowLayout

Os componentes são distribuídos da esquerda para direita, de cima para baixo, sendo que o tamanho dos componentes são ajustados individualmente.

• GridLayout

Os componentes são distribuídos em uma grade (tabela), da esquerda para direita, de cima para baixo, mas o tamanho dos componentes é ajustado no tamanho da célula da grade, ou seja, todos os componentes terão o mesmo tamanho.

• GridBagLayout

Similar ao GridLayout, porém as células da grade podem ter tamanhos diferentes. O tamanho de cada célula é definido pela classe **GridBagConstraints**.

• BorderLayout

Divide o retângulo do container em 5 partes: superior (north), inferior (South), esquerda (West), direita (East) e centro (Center). Ao adicionar cada componente, deve-se especificar uma destas posições.

• CardLayout

Utilizado para exibir um componente de cada vez.

• BoxLayout

Respeita o tamanho pré-definido dos componentes, alinhando-os em linhas ou colunas.

14


**Exemplo: usando layouts**

Para ilustrar o uso dos layouts vamos criar um formulário com uma planilha de cadastro (várias caixas de texto dentro de um **GridLayout**). Para isto, siga o roteiro abaixo:

- Primeiro, vamos definir que queremos uma tela que permita o preenchimento de 5 campos: nome, endereço, telefone, cidade e cep.
- Vamos criar um projeto **Ex5Layouts** e uma classe principal **ExemploGridLayout** para implementar este formulário.
- Vamos colocar 2 botões: OK e Cancelar. OK exibirá uma mensagem contendo tudo que foi escrito nos campos. Cancelar fechará a janela.
- Este formulário será chamado de outra janela, que será a janela principal do programa. Para isto, criaremos uma segunda classe, que implementará o formulário, chamada **FormCadastro**.

15

O código do **FormCadastro** ficará assim:

```
package modulo1;

import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
```

```
import javax.swing.JOptionPane;
import javax.swing.JTextField;

public class FormCadastro extends JFrame {

    private JButton butOK;
    private JButton butCancelar;
    private JTextField campoNome,campoEnd,campoTel,campoCid,campoCep;
    private JLabel textoNome,textoEnd,textoTel,textoCid,textoCep;

    public FormCadastro() {
        super("Cadastro de cliente");

        textoNome = new JLabel("Nome:"); campoNome = new JTextField(15);
        textoEnd = new JLabel("Endereço:"); campoEnd = new JTextField(15);
        textoTel = new JLabel("Fone:"); campoTel = new JTextField(15);
        textoCid = new JLabel("Cidade:"); campoCid = new JTextField(15);
        textoCep = new JLabel("CEP:"); campoCep = new JTextField(15);

        butOK = new JButton("OK"); butCancelar = new JButton("Cancelar");

        setLayout(new GridLayout(3,2));
        add(textoNome); add(campoNome);
        add(textoEnd); add(campoEnd);
        add(textoTel); add(campoTel);
        add(textoCid); add(campoCid);
        add(textoCep); add(campoCep);

        add(butOK); add(butCancelar);

        butOK.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                JOptionPane.showMessageDialog(null,
campoNome.getText()+";"+
campoEnd.getText()+";"+
campoTel.getText()+";"+
campoCid.getText()+";"+
campoCep.getText());
            }
        });

        butCancelar.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                setVisible(false);
                dispose();
            }
        });

        pack();
    }
}
```

Note que a função **add** do JFrame organiza os componentes a medida que são adicionados, conforme o layout escolhido previamente em**setLayout**.

**FormCadastro** é uma classe filha de **JFrame**, então, todos os métodos (public e protected) de JFrame podem ser chamados internamente nos métodos desta classe.

16

Já a classe do programa principal, que irá abrir a **FormCadastro**, terá o método **main** que inicia o programa:

```
package modulo1;

import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class ExemploGridLayout {

    /**
   * @param args
   */
    public static void main(String[] args) {
        JFrame frm = new JFrame("Exemplo de GridLayout");

        FlowLayout flow = new FlowLayout();
        frm.setLayout(flow);

        frm.setVisible(true);
        JLabel lbl = new JLabel("Aperte o botão abaixo para abrir o formulário");
        frm.add(lbl);

        JButton but = new JButton("Aperte aqui");
        frm.add(but);

        frm.setSize(350,200);
        frm.pack();

        frm.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
```

```
        but.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                FormCadastro form = new FormCadastro();
                form.setVisible(true);
            }
        });
    }
}
```

17

### 2.3 Look and Feel

**Look and Feel** é um recurso do swing que permite alterar os componentes de interface como um todo.

Para experimentá-lo basta adicionar o seguinte código na primeira linha do método **main** de seu programa:

```
UIManager.setLookAndFeel("nome da classe que implementa o look and feel");
```

Experimente, por exemplo, adicionar uma das linhas abaixo no exemplo anterior (FormCadastro), para verificar o que acontece:

```
UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
```

Ou

```
UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
```

Ou

```
UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
```

18

### 2.4 Menus

A maioria dos programas “Desktop” possuem menus. Basicamente há 2 tipos de menu:

• JMenuBar

Que implementa uma barra horizontal de menu, útil para utilizar na tela principal de um aplicativo. Um JMenuBar pode possuir vários submenus verticais, um para cada opção, que são implementados pela própria classe JMenu.

• JPopupMenu

Que implementa um menu popup, que normalmente é usado para menus **properties**, que são ativados pelo botão direito do mouse.

Para ilustrar a utilização dos menus, siga o roteiro abaixo:

- No Eclipse, vamos copiar o projeto **ex5Layouts** criando um novo projeto **ex6Menus**.
- Renomeie a classe **ExemploGridLayout** para **ExemploMenus**.
- Exclua a linha que adicionam o label e o button. Retire o método **pack**.
- Para criar o menu principal, use o código abaixo:

```
JMenuBar menuBar = new JMenuBar();
frm.setJMenuBar(menuBar);
```

- Para criar um submenu, use o código abaixo:

```
JMenu cadMenu = new JMenu("Cadastro");
menuBar.add(cadMenu);
```

- Para adicionar uma opção no submenu, use o código abaixo:

```
JMenuItem newAction = new JMenuItem("Novo cadastro");
cadMenu.add(newAction);
```

- Para adicionar o código a ser executado quando o usuário escolher a opção newAction, use o código abaixo:

```
   newAction.addActionListener(new ActionListener(){
           public void actionPerformed(ActionEvent e){
               FormCadastro form = new FormCadastro();
               form.setVisible(true);
           }
       });
```

19

Enfim, o código completo do **ExemploMenus** fica assim:

```
package modulo1;

import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;

public class ExemploMenus {

    /**
   * @param args
   * @throws UnsupportedLookAndFeelException
   * @throws IllegalAccessException
   * @throws InstantiationException
   * @throws ClassNotFoundException
   */
    public static void main(String[] args) throws ClassNotFoundException,
InstantiationException, IllegalAccessException, UnsupportedLookAndFeelException {
        UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");

        JFrame frm = new JFrame("Exemplo de Menus");

        FlowLayout flow = new FlowLayout();
        frm.setLayout(flow);

        frm.setVisible(true);

        // Cria uma barra de menu para o JFrame
            JMenuBar menuBar = new JMenuBar();

            // Adiciona a barra de menu ao  frame
            frm.setJMenuBar(menuBar);

            // Define e adiciona dois menus drop down na barra de menus
            JMenu cadMenu = new JMenu("Cadastro");
            menuBar.add(cadMenu);
            JMenu ajuMenu = new JMenu("Ajuda");
            menuBar.add(ajuMenu);

            // Cria e adiciona um item simples para o menu
            JMenuItem newAction = new JMenuItem("Novo cadastro");
            cadMenu.add(newAction);
            JMenuItem edAction = new JMenuItem("Editar cadastro");
            cadMenu.add(edAction);
            JMenuItem sairAction = new JMenuItem("Sair");
            cadMenu.addSeparator();
            cadMenu.add(sairAction);
            JMenuItem sobAction = new JMenuItem("Sobre");
         ajuMenu.add(sobAction);

        frm.setSize(650,200);
```

```
        //frm.pack();

        frm.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });

        newAction.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                FormCadastro form = new FormCadastro();
                form.setVisible(true);
            }
        });

        edAction.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                FormCadastro form = new FormCadastro();
                form.setVisible(true);
            }
        });

        sairAction.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                System.exit(0);
            }
        });

        sobAction.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                //FormCadastro form = new FormCadastro();
                //form.setVisible(true);
            }
        });
    }
}
```

20

**Que tal praticar?**

a) Adicione ao exemplo anterior, um outro submenu “Opções”, com 5 itens, na barra de menus.

b) Adicione um ActionListener em cada um destes itens, e mostre um JOptionPane para demonstrar qual opção foi clicada.

```
JOptionPane.showMessageDialog(null, “opção x”);
```

21

## 3 - Criando uma aplicação completa, com Swing / Awt

Recapitulando, até agora aprendemos:

- colocar componentes de interface num formulário e ajustá-los com um layout;
- criar menus;
- abrir um formulário a partir de uma tela principal;
- programar o que acontece quando um usuário realiza uma operação de interface (evento tal como clicar num botão);
- criar caixas de aviso (JOptionPane).

Agora iremos exemplificar como tornar tudo isto útil. Iremos utilizar estas habilidades básicas para criar uma aplicação simples, tal como um cadastro de clientes.

Para isso, é interessante organizarmos nosso projeto, adotando um padrão de projeto (design pattern), a fim de garantir organização enquanto o sistema cresce:

- Uma classe para cada formulário.
- Organizar as actions de maneira similar.
- Manter uma padronização do código fonte.
- Utilizar formulários modais (enquanto o usuário está acessando um formulário, não se deve acessar formulários que estão embaixo, a não ser que se queira esta funcionalidade).
- Separar o código-fonte relacionado às regras de negócio do relacionado a interface (veremos mais sobre isto no estudo do padrão MVC).

22

É muito comum no desenvolvimento de sistemas o programador “perder o fio da meada”, ou seja, criar um código tão complexo e desorganizado, que nem ele, o próprio criador, consegue realizar manutenção depois. Para se evitar isto, segue-se algumas regras básicas:

- Utilização dos recursos que uma linguagem orientada a objetos oferece, tais como herança, interfaces e polimorfismo. Por exemplo: você pode criar uma classe JFormularioEdit, que é genérica e serve para editar registros, e uma classe-filha JFormularioEditCliente, que edita registros de clientes.
- Códigos relacionados a um requisito não funcional (requisitos tecnológicos) devem ser separados dos códigos relacionados a requisitos funcionais (requisitos de regra de negócio). Por exemplo: para validar um CPF, evite colocar este código no próprio código-fonte de um formulário de edição de registro. Crie uma classe “validadores” e um método “validarCpf”.

- Criar códigos para serem reutilizados (genérico): toda vez que se identificar um código que possa ser reutilizado em outro local ou outros sistemas, organizar este código de forma que ele possa ser reutilizado no futuro.
- Não reinvente a roda: antes de se construir algo básico, verificar com o google se alguém já fez algo similar.

23

### 3.1 - Formulários Modais

Caixa de diálogo modal é um formulário que "tranca" todas as telas inferiores até que o usuário finalize a utilização deste formulário.

Para criarmos este tipo de tela, utilizamos o JDialog, da seguinte forma:

```
JDialog dlg = new JDialog(frameInferior, "", Dialog.ModalityType.DOCUMENT_MODAL);
```

No entanto, para organizar nosso código, podemos criar uma classe que implementa um formulário (por exemplo FormSobre). Esta classe pode implementar a "forma modal" de se comportar. Ela teria o seguinte código:

```
package modulo1;

import java.awt.Dialog;
import java.awt.FlowLayout;
import java.awt.Frame;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JLabel;

public class FormSobre extends JDialog {
        public FormSobre(Frame frm) {
                super(frm,"",Dialog.ModalityType.DOCUMENT_MODAL);

                FlowLayout flow = new FlowLayout();
                setLayout(flow);

                JLabel creditos = new JLabel("Cadastro de clientes 1.0");
                add(creditos);

                JButton butOK = new JButton("OK");
                add(butOK);

                setSize(250,200);

                butOK.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
```

```
                                setVisible(false);
                                dispose();
                            }
                        });
                }
        }
```

Note que a linha

```
super(frm,"",Dialog.ModalityType."DOCUMENT_MODAL");
```

chama o mesmo construtor descrito no início deste tópico.

24

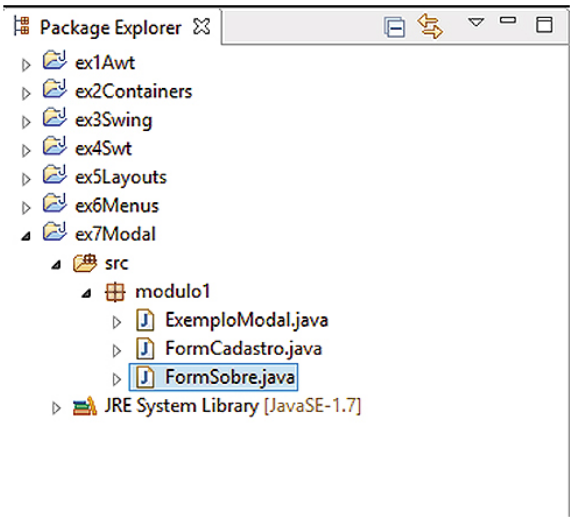
Para chamarmos este diálogo, a partir da tela principal do programa ExemploMenu, anteriormente visto, basta adicionar o seguinte action no corpo da classe ExemploMenu:

```
    sobAction.addActionListener(new ActionListener(){
                            public void actionPerformed(ActionEvent e){
                                    FormSobre sobreDlg = new FormSobre(frm);
                                    sobreDlg.setVisible(true);
                            }
                    });
```

Ao executar, note que enquanto o formulário FormSobre estiver aberto, o usuário não consegue colocar foco na tela principal.

Para organizar, copie todo o projeto ex6Menus em outro projeto ex7Modal, e renomeie a classe **ExemploMenu** para **ExemploModal**.

O projeto então deverá estar organizado da seguinte maneira no eclipse:



**Que tal praticar?**

Converta o formulário **FormCadastro** em modal, a partir da tela principal do ExemploModal.

25

### 3.2 - Listas

Para construir um exemplo de lista, vamos copiar o projeto ExemploModal para um novo projeto ExemploLista.

Iremos por uma lista na tela principal de nosso aplicativo, mostrando a lista de clientes. Para isso, vamos organizar nosso projeto ExemploLista com as seguintes tarefas:

- Renomear a classe ExemploModal para ExemploLista.
- Em ExemploLista, antes da linha frm.setSize(650,200); adicionar o seguinte código abaixo:

```
        JPanel topPanel = new JPanel();
                topPanel.setLayout( new BorderLayout() );
                frm.add( topPanel );
                DefaultListModel listModel = new DefaultListModel();
                        listModel.addElement("Fulano");
                        listModel.addElement("Beltrano");
                        listModel.addElement("Sicrano");
                JList listbox = new JList( listModel );
                topPanel.add( listbox, BorderLayout.CENTER );
```

Isso irá exibir uma lista na tela principal. Note que a classe **DefaultListModel** implementa uma lista em memória, e o JList exibe o que houver na instância dela, passada como parâmetro, no construtor do JList: listbox = new JList( listModel );

26

• Logo abaixo do main, da classe ExemploLista, altere o FlowLayout para BorderLayout, da seguinte forma:

```
BorderLayout flow = new BorderLayout();
frm.setLayout(flow);
```

Ao executar a classe ExemploLista, aparecerá a seguinte tela:


E se quisermos adicionar novos itens na lista?

Podemos utilizar a própria opção **Novo** do menu **Cadastro**. Para isso, vamos adicionar na lista o que for digitado no campo nome, de nossa tela **FormCadastro**.

Mas para isto, precisamos ter um meio de obter o texto digitado no campo **nome** e passarmos para a classe da tela principal **ExemploLista**. Vejamos a seguir.

27

Vamos então criar 2 métodos em FormCadastro: getResult() e getNome().

- getResult irá retornar verdadeiro se o usuário apertar OK, e neste caso, o nome do cliente deverá ser inserido na lista;
- getNome irá retornar o nome digitado no campo nome.

Então, o código do FormCadastro ficará assim:

```
package modulo1;

import java.awt.Dialog;
import java.awt.Frame;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JLabel;
import javax.swing.JTextField;

public class FormCadastro extends JDialog {

        private JButton butOK;
        private JButton butCancelar;
        private JTextField campoNome,campoEnd,campoTel,campoCid,campoCep;
        private JLabel textoNome,textoEnd,textoTel,textoCid,textoCep;
```

```
private boolean result = false;

        public FormCadastro(Frame frm) {
                super(frm, "Cadastro de cliente",
Dialog.ModalityType.DOCUMENT_MODAL);

                textoNome = new JLabel("Nome:"); campoNome = new JTextField(15);
                textoEnd = new JLabel("Endereço:"); campoEnd = new JTextField(15);
                textoTel = new JLabel("Fone:"); campoTel = new JTextField(15);
                textoCid = new JLabel("Cidade:"); campoCid = new JTextField(15);
                textoCep = new JLabel("CEP:"); campoCep = new JTextField(15);

                butOK = new JButton("OK"); butCancelar = new JButton("Cancelar");

                setLayout(new GridLayout(3,2));
                add(textoNome); add(campoNome);
                add(textoEnd); add(campoEnd);
                add(textoTel); add(campoTel);
                add(textoCid); add(campoCid);
                add(textoCep); add(campoCep);

                add(butOK); add(butCancelar);

                butOK.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                                setVisible(false);
                                dispose();
                                result = true;
                        }
                });

                butCancelar.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                                setVisible(false);
                                dispose();
                        }
                });

                pack();
        }

        public String getNome() {
                return campoNome.getText();
        }

        public boolean getResult() {
                return result;
        }
}
```

E o código do **ExemploLista** ficará assim:

```
package modulo1;
```

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import javax.swing.DefaultListModel;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;

public class ExemploLista {

        private static JFrame frm;
        private static JList listbox;
        private static DefaultListModel listModel;

        /**
         * @param args
         * @throws UnsupportedLookAndFeelException
         * @throws IllegalAccessException
         * @throws InstantiationException
         * @throws ClassNotFoundException
         */
        public static void main(String[] args) throws ClassNotFoundException,
InstantiationException,
        IllegalAccessException, UnsupportedLookAndFeelException {

        UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");

                frm = new JFrame("Exemplo de Menus");

                BorderLayout flow = new BorderLayout();
                frm.setLayout(flow);

                frm.setVisible(true);

                // Cria uma barra de menu para o JFrame
                JMenuBar menuBar = new JMenuBar();

                // Adiciona a barra de menu ao  frame
                frm.setJMenuBar(menuBar);

                // Define e adiciona dois menus drop down na barra de menus
                JMenu cadMenu = new JMenu("Cadastro");
                menuBar.add(cadMenu);
                JMenu ajuMenu = new JMenu("Ajuda");
                menuBar.add(ajuMenu);
```

```
// Cria e adiciona um item simples para o menu
JMenuItem newAction = new JMenuItem("Novo cadastro");
cadMenu.add(newAction);
JMenuItem edAction = new JMenuItem("Editar cadastro");
cadMenu.add(edAction);
JMenuItem sairAction = new JMenuItem("Sair");
cadMenu. addSeparator();
cadMenu.add(sairAction);
JMenuItem sobAction = new JMenuItem("Sobre");
ajuMenu.add(sobAction);

JPanel topPanel = new JPanel();
topPanel.setLayout( new BorderLayout() );
frm.add( topPanel );
listModel = new DefaultListModel();
listModel.addElement("Fulano");
listModel.addElement("Beltrano");
listModel.addElement("Sicrano");
listbox = new JList( listModel );
topPanel.add( listbox, BorderLayout.CENTER );

frm.setSize(650,200);

frm.addWindowListener(new WindowAdapter()
{
        public void windowClosing(WindowEvent e){
                System.exit(0);
        }
});

newAction.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
                FormCadastro form = new FormCadastro(frm);
                form.setVisible(true);
                if (form.getResult()) {
                        listModel.addElement(form.getNome());
                }
        }
});

edAction.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
                FormCadastro form = new FormCadastro(frm);
                form.setVisible(true);
        }
});

sairAction.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
                System.exit(0);
        }
});
```

```
                sobAction.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                                FormSobre sobreDlg = new FormSobre(frm);
                                sobreDlg.setVisible(true);
                        }
                });
        }
}
```

28

## 4 - APLICAÇÃO FINAL

Para construirmos uma aplicação completa, copiaremos primeiro o projeto **Modal** visto anteriormente em um novo projeto **Final**.

Renomeie a classe ExemploLista para **CadastroClientes**.

Apesar de não ser uma classe relacionada com a interface gráfica, para manter a organização, vamos criar uma classe que represente um cliente em memória. Chamaremos esta classe de **Cliente**.

A classe **Cliente** será filha da classe **Object**. Por quê? Porque poderemos adicioná-la no JList e implementar a exibição do cliente sobrescrevendo um método **toString** da classe Object.

A classe Cliente terá o seguinte código:

```
package modulo1;

public class Cliente extends Object {
    private String nome;
    private String endereco;
    private String fone;
    private String cidade;
    private String cep;

    public Cliente(String nome, String endereco, String fone, String cidade, String
cep) {
        this.nome = nome;
        this.endereco = endereco;
        this.fone = fone;
        this.cidade = cidade;
        this.cep = cep;
    }

    @Override
    public String toString() {
        return "Nome: "+nome.toString()+" End.: "+endereco+" Fone: "+fone;
    }
}
```

29

Com o cursor sobre o código de Cliente, no Eclipse, clique com o botão direito do mouse, e escolha as opções **Source – Generate Getters and Setters**. Com isso, serão gerados os métodos **set** e **get** para cada propriedade da classe **Cliente**, em seu código-fonte.

Enquanto o programa estiver sendo executado, nosso cadastro de clientes ficará em uma lista em memória, que é a mesma acessada pelo JList. Para carregar a lista inicial, substituímos o código de carga da JList pelo seguinte código, em **CadastroClientes**

```
listModel = new DefaultListModel();
        listModel.addElement(new Cliente("Fulano","rua tal nro 0","5555-
5555","Brasilia","71000-000"));
        listModel.addElement(new Cliente("Beltrano","rua til nro 1","6666-
6666","Brasilia","71000-000"));
        listModel.addElement(new Cliente("Sicrano","rua tul nro 2","7777-
7777","Brasilia","71000-000"));
    listbox = new JList( listModel );
```

A tela **FormCadastro** deve também ser alterada para criar ou editar a classe **Cliente**. Para isto, adaptaremos o construtor da**FormCadastro**, para receber uma instância de Cliente, que será editado nesta tela:

```
public FormCadastro(Frame frm, Cliente cliente)
```

Adicionaremos uma propriedade **cliente** que manterá o cliente que está sendo editado acessível para todo o escopo da classe FormCadastro:

```
private Cliente cliente;
```

De volta ao construtor, precisamos carregar os dados da instância **cliente** nos componentes JTextField.

```
this.cliente = cliente;
        campoNome.setText(cliente.getNome());
        campoEnd.setText(cliente.getEndereco());
        campoTel.setText(cliente.getFone());
        campoCid.setText(cliente.getCidade());
        campoCep.setText(cliente.getCep());
```

30

E o que acontecerá quando o usuário apertar OK? Precisamos carregar a instância **cliente** com os valores que foram digitados nos componentes JTextField. Assim modificamos o ActionListener do botão OK da seguinte forma:

```
butOK.addActionListener(new ActionListener(){
```

```
            public void actionPerformed(ActionEvent e){
                setVisible(false);
                dispose();

                getCliente().setNome(campoNome.getText());
                getCliente().setEndereco(campoEnd.getText());
                getCliente().setFone(campoTel.getText());
                getCliente().setCidade(campoCid.getText());
                getCliente().setCep(campoCep.getText());

                result = true;
            }
    });
```

Enfim, precisamos devolver a instância de **cliente** preenchida para a **CadastroCliente**. Para isto, implementamos o método abaixo em**FormCadastro**:

```
public Cliente getCliente() {
        return cliente;
    }
```

Agora, a classe **CadastroCliente** precisa ser corrigida para trabalhar com o novo construtor de **FormCadastro** e com o método **getCliente**. Teremos que corrigir então o código relacionado com as opções **Novo** e **Editar** do menu:

```
   newAction.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                FormCadastro form = new FormCadastro(frm, new Cliente("<Novo
cliente>","","","","",""));
                form.setVisible(true);
                if (form.getResult()) {
                    listModel.addElement(form.getCliente());
                }
            }
        });

        edAction.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){

                int sel = listBox.getSelectedIndex();

                Cliente cliente = (Cliente) listModel.elementAt(sel);

                FormCadastro form = new FormCadastro(frm, cliente);
                form.setVisible(true);

                if (form.getResult()) {
                    listModel.setElementAt(form.getCliente(), sel);
                }
            }
        });
```

31

## 5 - Coesão

Vejamos o exemplo de classe a seguir:

```
class cliente{
    private nome;
    private telefone;
    private endereço;
    public cliente(String nome){
...
}
    public void setTelefone(String telefone){
...
}
public String getTelefone(){
...
}
public setEndereco(String telefone){
...
}
public String getEndereco(){
...
}
Public void imprimeCliente(){
limpaTela();
…

}
public void limpaTela(){
...
}
}
```

Podemos verificar que a classe tem um método chamado **limpaTela()**, que não tem relação com a proposta da classe cliente que é a representação de um cliente. O processamento de tela não deveria estar na classe, pois ela pode ser dependente do dispositivo. Então, caso quiséssemos alterar o dispositivo de apresentação, teríamos que alterar esse método que está dentro da classe cliente. Imagina quanto tempo perderíamos procurando na aplicação todos os métodos de impressão para fazer essa alteração.

32

Uma forma de resolver isso é criar uma classe específica para manipular a tela, conforme segue:

```
class tela{
    public void limpaTela(){
            ...
    }
}
```

E, em seguida, alterar o código da classe cliente para considerar essa alteração:

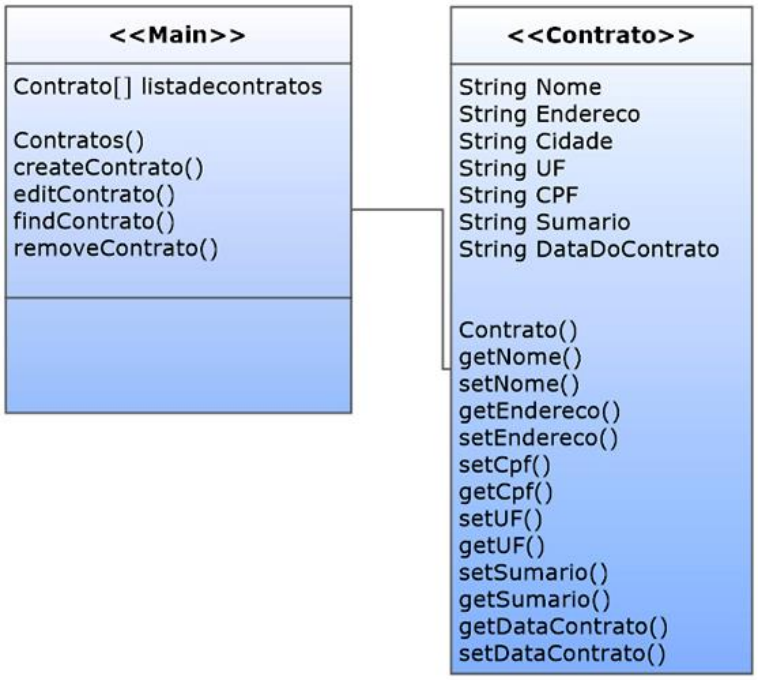
```
class cliente{
    private nome;
    private telefone;
    private endereço;
    public cliente(String nome){
    ...
    }
    public void setTelefone(String telefone){
    ...
    }
    public String getTelefone(){
    ...
    }
    public setEndereco(String telefone){
            ...
        }
        public String getEndereco(){
            ...
        }
        public void imprimeCliente(tela screen){
    screen. limpaTela();
             …
        }
}
```

Podemos ver no exemplo acima, que cada classe tem uma única responsabilidade, melhorando a organização do código e sua manutenção.

33

O problema maior é identificar como implementar a coesão na prática. Por exemplo, qual seria a melhor forma de implementar as classes de um sistema que possibilite a gestão de contratos?

Modelo inicial:

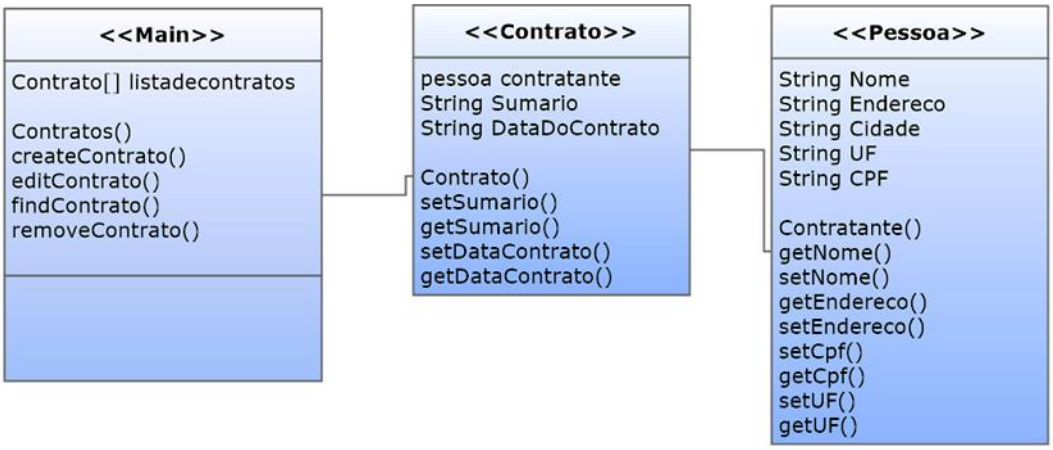



Podemos ver que a classe **contrato** possui baixa coesão, pois ela contém atributos que não estão diretamente relacionados a um contrato, como os dados pessoais do contratante.

34

Uma forma de melhorar então a coesão desse modelo seria separar as informações relativas ao contratante em uma classe específica.

Modelo ajustado:




O modelo ajustado segrega claramente a responsabilidade da classe contrato da responsabilidade de cadastro do contratante.

35

## 6 - ACOPLAMENTO

No item anterior pudemos ver que uma clara separação de responsabilidades contribui para um design organizado, fácil de ser mantido e alterado. A classe cliente não assume o papel de limpar a tela, ela deve pedir a quem tem a responsabilidade para que faça tal tarefa. A classe contratante não é mais encarregada de cadastrar o contratante, pois criamos uma classe específica para tal.

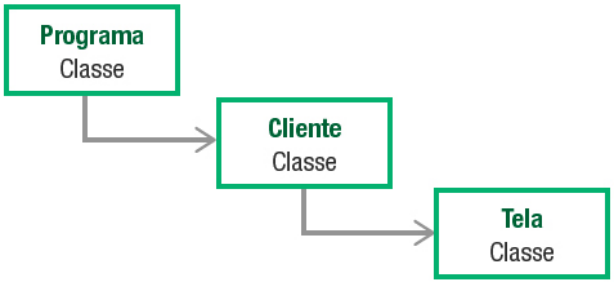
O que é importante ser considerado em relação à coesão, é que uma classe deve ser responsável por exercer uma única responsabilidade e fazer outras classes cooperarem quando necessário.

Já o **acoplamento** está relacionado à dependência de uma classe em relação à outra para funcionar. Quanto maior for esta dependência entre ambas, dizemos que estas classes elas estão fortemente acopladas.

Diversos problemas estão relacionados ao forte acoplamento, sendo que partes desses são muito similares àqueles de um ambiente com classes pouco coesas.

36

Vamos retornar ao nosso exemplo anterior, que foi alterado para aumentar a coesão da classe cliente. Nesse exemplo, separamos as responsabilidades, mas acabamos criando uma dependência muito forte da classe cliente com a classe tela, visto que o método de impressão deve necessariamente receber um objeto do tipo **tela**.



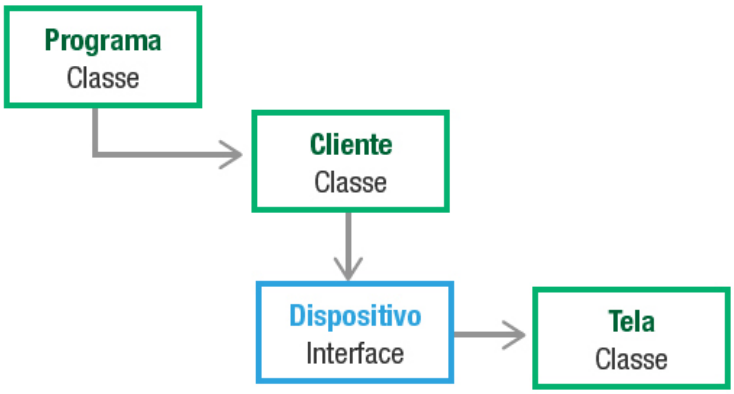

Isso indica um forte acoplamento da classe Cliente com a classe tela, o que não é desejável. **Então o que fazer quando nos deparamos com um projeto que não tenha essas características?**

A solução é a refatoração que consiste em rever a arquitetura da aplicação de forma a buscar reduzir o acoplamento entre as classes e aumentar a coesão. Uma das formas mais comuns de reduzir o acoplamento entre as classes é usar interfaces.

37

Vimos que uma interface pode ser vista como um protocolo, de forma que todos as classes que implementarem a interface atenderão aos requisitos mínimos necessários para realizar uma determinada funcionalidade.

No exemplo do cliente, poderíamos fazer a seguinte alteração no modelo:




Que seria obtido alterando-se o método imprimeCliente(), como segue:

```
Public void imprimeCliente(dispositivo screen){
        screen. limpaTela();
        …
}
```

E a interface dispositivo poderia ser declarada como segue:

```
public interface dispositivo{
        public void limpaTela();
}
```

Por fim, teríamos que alterar a declaração da classe tela de forma a implementar a interface dispositivo:

```
class tela implements dispositivo{
       public void limpaTela(){
           ...
       }
}
```

Podemos perceber que agora não existe mais a dependência da classe cliente com a classe tela. Na verdade, agora o método**imprimeCliente** poderá receber qualquer tipo de objeto que implemente a interface dispositivo. Essa solução aumenta a flexibilidade e usabilidade dessa classe.

38

## RESUMO

Neste módulo aprendemos a criar uma aplicação gráfica completa com Java usando as principais bibliotecas disponíveis: AWT, Swing e SWT. Vimos que existem basicamente dois tipos de elementos de interface: containers e atômicos. Os elementos do tipo container permitem a inclusão de outros elementos em seu interior. São exemplos de containers: JPanel, JFrame e JApplet. Os elementos atômicos são elementos básicos que não admitem nenhum elemento de interface em seu interior, como JButton, JLabel, JTextField e JScrollBar.

Vimos ainda neste módulo como responder a eventos de interface, como criar menus e trabalhar com formulários e listas em ambientes gráficos. E foi apresentada uma aplicação gráfica completa que poderá servir de referência para futuros desenvolvimentos.

Por fim, tratamos de dois conceitos de programação orientada a objetos muito importantes: acoplamento e coesão. Sabemos que no desenvolvimento de aplicações reais nem sempre encontramos um cenário ideal de aplicações com classes com um baixo acoplamento e uma alta coesão. Mas como futuros arquitetos, temos que buscar sempre a melhoria dos processos, melhoria do modelo usado no desenvolvimento dos sistemas. A compreensão dos conceitos de coesão e acoplamento são fundamentais para as boas práticas de desenvolvimento de *software*. A coesão está relacionada com a definição clara do escopo da classe e de sua responsabilidade. Quanto mais bem definido estiver o escopo da classe, quanto maior a sua especialização maior será a coesão da classe. O acoplamento, por sua vez, busca avaliar o grau de dependência da classe em relação às outras. Isso é importante, porque classes com acopladas fortemente são mais difíceis de serem mantidas de forma autônoma e, provavelmente, quando tivermos que alterar uma classe teremos que alterar a outra. Esses dois princípios permitem nos ajudar na tomada de decisões durante o desenvolvimento da arquitetura e a disciplina na aplicação dos dois conceitos garantirá uma maior facilidade de desenvolvimento e manutenção das aplicações.